made with doc-O-matic

RAD Studio (Common)

Table of Contents

Concepts 1

Compiling, Building, and Running Applications 2

Compiling, Building, and Running Applications 2

MSBuild Overview 4

Build Configurations Overview (Delphi) 5

Build Configurations Overview (C++) 6

Named Option Sets Overview 7

Targets files 8

Debugging Applications 10

Overview of Debugging 10

Overview of Remote Debugging 12

General Concepts 14

Managing the Development Cycle Overview 14

Designing User Interfaces 15

Using Source Control 16

Localizing Applications 18

Deploying Applications 19

Getting Started 22

What is RAD Studio? 23

What's New in RAD Studio (Delphi for Microsoft .NET) 24

What's New in RAD Studio (C++Builder) 28

What's New in RAD Studio (Delphi) 32

Tour of the IDE 34

IDE on Windows Vista 40

Tools Overview 41

Code Editor 42

Form Designer 46

Starting a Project 47

Template Libraries 50

Overview of Virtual Folders 50

Help on Help 51

Code Completion 53

Refactoring Applications 55

Add Namespace 57

Refactoring Overview 57

Change Parameters 58

Symbol Rename Overview (Delphi, C#, C++) 59

Add or Edit Parameter 60

Extract Method Overview (Delphi) 60

Declare Field 61

Extract Resource String (Delphi) 62

Declare Variable and Declare Field Overview (Delphi) 62

Declare Variable 64

Extract Method 64

Find References Overview (Delphi, C#, C++) 65

Change Parameters Overview (Delphi) 66

Extract Resource String 66

Find Unit 66

Sync Edit Mode (Delphi, C#, C++) 67

Refactorings 67

Undoing a Refactoring (Delphi, C#) 67

Rename Symbol (C++) 68

Rename <symbol name> (C#) 68

Rename <symbol name> (Delphi) 69

Testing Applications **70**

Unit Testing Overview 70

DUnit Overview 72

NUnit Overview 76

Modeling Applications with Together **81**

Getting Started with Together 83

About Together 83

UML 2.0 Sample Project 84

Code Visualization Overview 87

What's New in Together 88

Modeling Overview 89

Together Project Overview 89

Namespace and Package Overview 89

Together Diagram Overview 90

Supported UML Specifications 90

Model Element Overview 91

Model Annotation Overview 91

Model Shortcut Overview 92

Diagram Layout Overview 92
Model Hyperlinking Overview 92
LiveSource Overview 93
Transformation to Source Code Overview 94
OCL Support Overview 95
Patterns Overview 96
Refactoring Overview 98
Quality Assurance Facilities Overview 98
Documentation Generation Facility Overview 100
Model Import and Export Overview 100

Procedures 103

Compiling and Building Procedures 104
Applying the Active Build Configuration for a Project 104
Building Packages 105
Compiling C++ Design-Time Packages That Contain Delphi Source 106
Creating Build Events 107
Creating Named Build Configurations for C++ 107
Creating Named Build Configurations for Delphi 108
Building a Project Using an MSBuild Command 108
Using Targets Files 109
Installing More Computer Languages 110
Linking Delphi Units Into an Application 111
Previewing and Applying Refactoring Operations 111
Renaming a Symbol 112
Working with Named Option Sets 113

Debugging Procedures 114
Adding a Watch 116
Using the CPU View 116
Displaying Expanded Watch Information 117
Attaching to a Running Process 117
Setting and Modifying Breakpoints 118
Debugging VCL for .NET Source Code 121
Using Tooltips During Debugging 122
Inspecting and Changing the Value of Data Elements 122
Modifying Variable Expressions 124
Preparing a Project for Debugging 124
Remote Debugging: Metaprocedure 125

Installing, Starting, and Stopping the Remote Debug Server 125
Installing a Debugger on a Remote Machine 126
Establishing a Connection for Remote Debugging 127
Preparing Files for Remote Debugging 128
Setting the Search Order for Debug Symbol Tables 129
Resolving Internal Errors 130

Deploying Applications 133
Deploying ASP.NET applications 133
Deploying the AdoDbx Client 134

Editing Code Procedures 136
Using Code Folding 137
Creating Live Templates 138
Creating Template Libraries 138
Customizing Code Editor 141
Finding References 141
Finding Units and Using Namespaces (Delphi, C#) 142
Recording a Keystroke Macro 142
Refactoring Code 143
Using Bookmarks 145
Using Class Completion 145
Using Code Insight 146
Using Live Templates 148
Using the History Manager 149
Using Sync Edit 150

Getting Started Procedures 151
Adding Components to a Form 152
Adding References 153
Adding and Removing Files 153
Adding Templates to the Object Repository 153
Copying References to a Local Path 154
Creating a Component Template 154
Creating a Project 155
Customizing the Form 155
Customizing the Tool Palette 156
Customizing Toolbars 156
Disabling Themes in the IDE and in Your Application 157
Docking Tool Windows 157
Finding Items on the Tool Palette 158

Exploring .NET Assembly Metadata Using the Reflection Viewer 158
Exploring Windows Type Libraries 159
Installing Custom Components 160
Renaming Files Using the Project Manager 160
Saving Desktop Layouts 161
Setting Component Properties 161
Setting Dynamic Properties 161
Setting Project Options 162
Setting C++ Project Options 163
Setting Properties and Events 164
Setting The IDE To Mimic Delphi 7 164
Setting Tool Preferences 165
Using Design Guidelines with VCL Components 165
Using the File Browser 166
Using To-Do Lists 166
Using Virtual Folders 167
Writing Event Handlers 168

Localization Procedures 169
Adding Languages to a Project 169
Editing Resource Files in the Translation Manager 170
Setting the Active Language for a Project 171
Setting Up the External Translation Manager 172
Updating Resource Modules 173
Using the External Translation Manager 173

Managing Memory 175
Configuring the Memory Manager 175
Increasing the Memory Address Space 176
Monitoring Memory Usage 177
Registering Memory Leaks 178
Sharing Memory 178

Unit Test Procedures 180
Developing Tests 180

Together Procedures 183
Configuring Together 183
Together Refactoring Procedures 184
Refactoring: Changing Parameters 184
Refactoring: Extracting Interfaces 185

Refactoring: Extracting Method 185

Refactoring: Extracting Superclass 186

Refactoring: Creating Inline Variables 186

Refactoring: Introducing Fields 187

Refactoring: Introducing Variables 187

Refactoring: Moving Members 188

Refactoring: “Pull Members Up" and “Push Members Down” 188

Refactoring: Renaming Elements 189

Refactoring: "Safe Delete" 189

Opening the UML 2.0 Sample Project 190

Together Diagram Procedures 190

Annotating a Diagram 195

Creating a Diagram 196

Exporting a Diagram to an Image 197

Printing a Diagram 197

Changing Diagram Notation 197

Using Grid and Other Appearance Options 198

Using the UML in Color Profile 198

Aligning Model Elements 199

Changing Type of a Link 199

Closing a Diagram 200

Copying and Pasting Model Elements 200

Deleting a Diagram 200

Hyperlinking Diagrams 201

Laying Out a Diagram Automatically 202

Moving Model Elements 203

Renaming a Diagram 204

Rerouting a Link 204

Resizing Model Elements 204

Selecting Model Elements 205

Assigning an Element Stereotype 205

Using Drag-and-Drop 206

Working with User Properties 206

Creating a Link with Bending Points 207

Creating Multiple Elements 207

Creating a Shortcut 208

Creating a Simple Link 209

Creating a Single Model Element 209

Searching Diagrams 209

Searching Source Code for Usages 210
Creating an Activity for a State 211
Designing a UML 1.5 Activity Diagram 211
Instantiating a Classifier 211
Designing a UML 1.5 Component Diagram 212
Designing a UML 1.5 Deployment Diagram 212
Adding a Conditional Block 213
Associating an Object with a Classifier 214
Branching Message Links 215
Converting Between UML 1.5 Sequence and Collaboration Diagrams 215
Working with a UML 1.5 Message 215
Designing a UML 1.5 Statechart Diagram 217
Creating a Pin 217
Designing a UML 2.0 Activity Diagram 218
Grouping Actions into an Activity 219
Working with an Object Flow or a Control Flow 219
Designing a UML 2.0 Component Diagram 220
Creating a Delegation Connector 221
Creating an Internal Structure for a Node 221
Creating a Referenced Part 221
Creating a Port 222
Working with a Collaboration Use 222
Designing a UML 2.0 Deployment Diagram 223
Associating a Lifeline with a Classifier 224
Copying and Pasting an Execution or Invocation Specification 224
Creating a Sequence or Communication Diagram from an Interaction 225
Creating a State Invariant 225
Designing a UML 2.0 Sequence or Communication Diagram 226
Linking Another Interaction from an Interaction Diagram 227
Working with a UML 2.0 Message 227
Working with a Combined Fragment 228
Working with a Tie Frame 229
Associating a Transition or a State with an Activity 230
Creating a Guard Condition for a Transition 230
Creating a History Element 230
Creating a Member for a State 231
Creating a State 231
Designing a UML 2.0 State Machine Diagram 232
Browsing a Diagram with Overview Pane 232

Hiding and Showing Model Elements 232
Using View Filters 233
Zooming a Diagram 234
Working with a Complex State 234
Creating a Deferred Event 235
Creating an Internal Transition 235
Creating a Multiple Transition 235
Creating a Self-Transition 236
Specifying Entry and Exit Actions 236
Working with an Instance Specification 237
Working with a Provided or Required Interface 238
Creating an Association Class 239
Creating an Inner Classifier 239
Using a Class Diagram as a View 240
Working with an Interface 240
Working with a Relationship 241
Adding a Member to a Container 241
Changing Appearance of Compartments 242
Changing Appearance of Interfaces 242
Working with a Constructor 243
Working with a Field 243
Associating a Message Link with a Method 244
Generating an Incremental Sequence Diagram 245
Creating a Browse-Through Sequence of Diagrams 246
Creating an Extension Point 246
Designing Use Case Hierarchy 246
Together Documentation Generation Procedures 247
Configuring the Documentation Generation Facility 247
Generating Project Documentation 248
Using Online Help 248
Together Object Constraint Language (OCL) Procedures 248
Creating an OCL Constraint 249
Editing an OCL Expression 249
Showing and Hiding an OCL Constraint 250
Working with a Namespace or a Package 250
Together Pattern Procedures 251
Adding Participants to the Patterns as First Class Citizens 253
Creating a Pattern 253
Deleting Patterns as First Class Citizens from the Model 254

Using the Pattern Registry 254

Creating a Link by Pattern 255

Creating a Model Element by Pattern 255

Using the Stub Implementation Pattern 255

Exporting a Pattern 257

Importing a Legacy Pattern 257

Sharing Patterns 258

Assigning Patterns to Shortcuts 258

Copying and Pasting Shortcuts, Folders or Pattern Trees 258

Creating a Folder in the Pattern Organizer 259

Creating a Shortcut to a Pattern 259

Creating a Virtual Pattern Tree 259

Deleting shortcuts, folders or pattern trees 260

Editing Properties 260

Opening the Pattern Organizer 260

Saving Changes in the Pattern Registry 261

Sorting Patterns 261

Using the Pattern Organizer 261

Together Project Procedures 262

Activating Together Support for Projects 263

Creating a Project 264

Exporting a Project to XMI Format 264

Importing a Project in IBM Rational Rose (MDL) Format 265

Importing a Project Created in TVS, TEC, TJB, or TPT 265

Importing a Project in XMI Format 266

Opening an Existing Project for Modeling 267

Synchronizing the Model View, Diagram View, and Source Code 267

Transforming a Design Project to Source Code 269

Troubleshooting a Model 269

Working with a Referenced Project 270

Together Quality Assurance Procedures 271

Exporting Audit Results 271

Printing Audit Results 272

Running Audits 273

Viewing Audit Results 274

Working with a Set of Audits 274

Creating a Metrics Chart 275

Running Metrics 276

Viewing Metric Results 276

Working with a Set of Metrics 277

Reference 279

Delphi Reference **280**

Delphi Compiler Directives (List) 280

Delphi compiler directives 282

Align fields (Delphi) 283

Application type (Delphi) 283

Assert directives (Delphi) 284

Autoboxing (Delphi for .NET) 284

Boolean short-circuit evaluation (Delphi compiler directive) 285

Conditional compilation (Delphi) 285

Debug information (Delphi) 287

DEFINE directive (Delphi) 287

DENYPACKAGEUNIT directive (Delphi) 287

Description (Delphi) 288

DESIGNONLY directive (Delphi) 288

ELSE (Delphi) 288

ELSEIF (Delphi) 289

ENDIF directive 289

Executable extension (Delphi) 289

Export symbols (Delphi) 290

Extended syntax (Delphi) 290

External Symbols (Delphi) 290

Floating Point Exception Checking (Delphi) 291

Hints (Delphi) 291

HPP emit (Delphi) 292

IFDEF directive (Delphi) 292

IF directive (Delphi) 292

IFEND directive (Delphi) 293

IFNDEF directive (Delphi) 294

IFOPT directive (Delphi) 294

Image base address 294

Implicit Build (Delphi) 295

Imported data 295

Include file (Delphi) 295

Input output checking (Delphi) 296

Compiler directives for libraries or shared objects (Delphi) 296

Link object file (Delphi) 297

Local symbol information (Delphi) 297

Long strings (Delphi) 298

Memory allocation sizes (Delphi) 298

MESSAGE directive (Delphi) 299

METHODINFO directive (Delphi) 299

Minimum enumeration size (Delphi) 299

Open String Parameters (Delphi) 300

Optimization (Delphi) 300

Overflow checking (Delphi) 301

Pentium-safe FDIV operations (Delphi) 301

NODEFINE 302

NOINCLUDE (Delphi) 302

Range checking 302

Real48 compatibility (Delphi) 302

Regions (Delphi and C#) 303

Resource file (Delphi) 303

RUNONLY directive (Delphi) 304

Runtime type information (Delphi) 304

Symbol declaration and cross-reference information (Delphi) 305

Type-checked pointers (Delphi) 305

UNDEF directive (Delphi) 306

Unsafe Code (Delphi for .NET) 306

Var-string checking (Delphi) 306

Warning messages (Delphi) 307

Warnings (Delphi) 308

Weak packaging 308

Stack frames (Delphi) 309

Writeable typed constants (Delphi) 309

PE (portable executable) header flags (Delphi) 310

Reserved address space for resources (Delphi) 310

Delphi Compiler Errors 311

Error Messages 311

Delphi Runtime Errors 509

I/O Errors 510

Fatal errors 511

Operating system errors 512

Delphi Language Guide 512

Classes and Objects 513

Data Types, Variables, and Constants 552

.NET Topics 593

Generics (Parameterized Types) 595

Inline Assembly Code (Win32 Only) 609

Object Interfaces 624

Libraries and Packages 634

Memory Management 644

Delphi Overview 656

Procedures and Functions 662

Program Control 678

Programs and Units 682

Standard Routines and I/O 692

Fundamental Syntactic Elements 700

RAD Studio Dialogs and Commands 730

Code Visualization 730

Code Visualization Diagram 730

Export Diagram to Image 731

Components 732

Create Component Template 732

Import Component 733

Packages 734

Assembly Search Paths 734

Installed .NET Components 734

.NET VCL Components 735

New Component 735

New VCL Component Wizard 736

Database 737

Add Fields 739

Assign Local Data 739

Columns Collection Editor 740

Constraints Collection Editor 740

Relations Collection Editor 740

Tables Collection Editor 741

CommandText Editor 741

Command Text Editor 742

Configure Data Adapter 742

Connection Editor 743

Connection Editor 743

Connection String Editor (ADO) 744

Data Adapter Dataset 744
DataAdapter Preview 744
Database Editor 745
Database Form Wizard 745
Dataset Properties 746
Driver Settings 746
Field Link Designer 746
Fields Editor 746
Foreign Key Constraint 747
Generate Dataset 748
New Connection 748
New Field 748
Relation 749
Rename Connection 749
SQL Monitor 749
Sort Fields Editor 750
Stored Procedures Dialog 751
TableMappings Collection Editor 751
Unique Constraint 752
IBDatabase Editor dialog box 752
IBTransaction Editor dialog box 753
IBUpdateSQL and IBDataSet Editor dialog box 753
Edit 754
Alignment 756
Creation Order 756
Edit Tab Order 756
Scale 757
Size 757
Align to Grid 757
Bring to Front 757
Copy 758
Cut 758
Delete 758
Flip Children 758
Lock Controls 758
Paste 758
Select All 759
Send to Back 759
Undo 759

Redo 759

Select All Controls 759

Error Messages 760

Data Breakpoint is set on a stack location 762

Misaligned Data Breakpoint 762

Error address not found 762

Another file named <FileName> is already on the search path 763

Could not stop due to hard mode 763

Error creating process: <Process> (<ErrorCode>) 763

A component class named <name> already exists 763

A field or method named <name> already exists 763

The project already contains a form or module named <Name> 763

Incorrect field declaration in class <ClassName> 763

Field <Field Name> does not have a corresponding component. Remove the declaration? 764

Field <Field Name> should be of type <Type1> but is declared as <Type2>. Correct the declaration? 764

Declaration of class <ClassName> is missing or incorrect 764

Module header is missing or incorrect 764

IMPLEMENTATION part is missing or incorrect 765

Insufficient memory to run 765

Breakpoint is set on line that contains no code or debug information. Run anyway? 765

<IDname> is not a valid identifier 765

<Library Name>is already loaded, probably as a result of an incorrect program termination. Your system may be unstable and you should exit and restart Windows now. 765

Incorrect method declaration in class <ClassName> 765

Cannot find implementation of method <MethodName> 766

The <Method Name> method referenced by <Form Name>.<Event Name> has an incompatible parameter list. Remove the reference? 766

The <Method Name> method referenced by <Form Name> does not exist. Remove the reference? 766

No code was generated for the current line 766

Property and method <MethodName> are not compatible 767

Cannot find <FileName.PAS> or <FileName.DCU> on the current search path 767

Source has been modified. Rebuild? 767

Symbol <BrowseSymbol> not found. 767

Debug session in progress. Terminate? 767

Uses clause is missing or incorrect 767

Invalid event profile <Name> 768

File 768

Active Form Wizard 771

Active Server Object wizard 772

Add 773

Automation Object Wizard 773
Browse With Dialog box 774
Browse With Dialog box 775
COM Object Wizard 775
COM+ Event Interface Selection dialog box 776
COM+ Event Object Wizard 776
COM+ Subscription Object Wizard 776
Customize New Menu 777
Change Destination File Name 777
FTP Connection Options 778
Interface Selection Wizard 778
New ASP.NET Application 779
New ASP.NET Content Page 779
New ASP.NET Generic Handler 779
New ASP.NET Master Page 780
New ASP.NET Master Page 780
New ASP.NET User Control 780
New ASP.NET Web Service 780
New Console Application 780
New DBWeb Control Wizard 781
New Dynamic-link Library 781
New Items 781
New Application 782
New Remote Data Module Object 782
New Thread Object 783
Open 784
Package 784
Print Selection 785
Project Upgrade 785
Project Updated 785
Remote Data Module Wizard 786
Satellite Assembly Wizard 787
Revert to Previous Revision 787
Add New WebService 787
SOAP Data Module Wizard 788
New SOAP Server Application 788
Save As 788
Select Directory 789
Transactional Object Wizard 789

Use Unit 790
WSDL Import Options 791
WSDL Import Wizard 792
New Web Server Application 793
Add New WebService 793
Application Module Page Options/New WebSnap Page Module 794
New WebSnap Application 795
New WebSnap Data Module 796
Web App Components 796
XML Data Binding Wizard Options 797
XML Data Binding Wizard, page 1 797
XML Data Binding Wizard, page 2 798
XML Data Binding Wizard, page 3 799
Close 799
Exit 799
New 800
HTML Elements 801
A (Anchor) HTML Element 801
Unit 803
DIV HTML Element 803
HR HTML Element 804
IMG HTML Element 805
INPUT HTML Element 806
SELECT HTML Element 808
SPAN HTML Element 809
TABLE HTML Element 810
TEXTAREA HTML Element 811
Insert 813
Insert User Control 813
Insert Image 813
Insert Input 814
Insert Table 814
Color Selector 815
Testing Wizards 815
Unit Test Case Wizard 816
Unit Test Case Wizard 816
Unit Test Project Wizard 816
Unit Test Project Wizard 817
NET_VS 817

Advanced Data Binding 818
AutoFormat 818
Collection Editor 818
Databindings 818
Dynamic Properties 818
Properties 819
Project 819
ASP.NET Deployment Manager 821
Project Options 822
COM Imports 846
C++ Project Options 847
.NET Assemblies 900
Project References 901
Add to Repository 901
UDDI Browser 902
Change Package 902
Project Dependencies 903
Add Languages 903
Remove Language 903
Set Active Language 903
New Category Name 904
Information 904
Project Page Options 904
Remove from Project 904
Options 905
Select Icon 906
Web Deploy Options 906
Build All Projects 907
Build Project 908
Compile and Make All Projects 908
Add to Project 908
Add New Project 909
Clean Package 909
Default Options 909
Options 909
Syntax Check for Project 910
Update Localized Projects 910
View Source 910
Propeditors 910

Delete Templates 914

Insert Template 915

Select Menu 915

Browse dialog box 915

Change Icon dialog box 916

Color editor 917

DDE Info dialog box 918

Filter editor 918

Font editor 919

Action Manager editor 919

Action List editor 921

Add Page dialog box 922

Collection Editor 922

Edit Page dialog box 923

IconView Items editor 924

Image List Editor 925

ListView Items Editor 926

New Standard Action Classes dialog box 927

String List editor 927

TreeView Items Editor 927

Value List editor 928

Input Mask editor 929

Insert Object dialog box 930

Loading an image at design time 931

Load Picture dialog box 931

Load String List dialog box 932

Masked Text editor 932

Notebook editor 933

Open dialog box 933

Paste Special dialog box 934

Picture editor 934

Save Picture As dialog box 935

Save String List dialog box 936

Run 936

Add Address Breakpoint or Add Data Breakpoint 938

Add Source Breakpoint 940

Attach to Process 941

Change 941

Debug Inspector 942

Debugger Exception Notification 943
Evaluate/Modify 944
Find Package Import 944
Inspect 945
Load Process Environment Block 945
Load Process Local 946
Load Process Remote 946
Load Process Symbol Tables 947
New Expression 948
Debug session in progress. Terminate? 948
Type Cast 948
Watch Properties 948
Detach From Program 949
Load Process 949
Parameters 950
Program Reset 950
Register ActiveX Server 950
Run 950
Run To Cursor and Run Until Return 950
Show Execution Point 951
Step Over 951
Trace Into 951
Trace to Next Source Line 951
Unregister ActiveX Server 951
Search 952
Find 952
Find in Files 953
Find References 954
Enter Address to Position 954
Go to Line Number 954
Replace Text 955
Search Again 955
Find Class 955
Find Local References 956
Find Original Symbol 956
Find References 956
Incremental Search 956
Together 957
Add New Diagram dialog box 960

Add/Remove Parameters for Operation dialog box 961
Add/Remove User Properties dialog box 961
Change Parameters dialog box 961
Choose Destination (or: Source) dialog box 962
Diagram Layout Algorithms 963
Edit Hyperlinks for Diagram dialog box 963
Export Diagram to Image dialog box 964
Extract Interface or Superclass dialog box 964
Extract Method dialog box 965
Generate Documentation dialog box 966
Generate Sequence Diagram dialog box 966
Inline Variable dialog box 967
Introduce Field dialog box 968
Introduce Variable dialog box 968
Model Support 969
Move dialog box 969
Together Options dialog window 969
Print Audit dialog box 970
Print Diagram dialog box 971
Pull Members Up and Push Members Down dialog boxes 972
QA Audits dialog window 972
QA Metrics dialog window 974
Rename 975
Safe Delete dialog box 975
Save Audit and Metric Results dialog box 976
Search for Usages dialog box 977
Select element dialog box 977
Selection Manager 978
XMI Export dialog box 978
XMI Import dialog box 979
Tools 979
CodeGuard Configuration 980
Tools Options 982
Configure Tools 1008
Edit Object Info 1008
Edit Tools 1008
Export Visual Studio Project 1009
History Manager 1009
Object Repository 1010

Template Libraries 1010
Tools Properties 1011
XML Mapper 1011
Web App Debugger 1013
View 1014
Add to Repository 1017
Debug Windows 1018
Code Explorer 1033
Customize Toolbars 1033
Data Explorer 1034
Delete Saved Desktop 1036
Desktop Toolbar 1036
File Browser 1036
Add to Repository 1037
Message View 1038
Object Inspector 1038
Project Manager 1038
Save Desktop 1047
Select Debug Desktop 1047
Structure View 1047
Templates Window 1048
To-Do List 1049
Add or Edit To-Do Item 1049
Filter To-Do List 1050
Table Properties 1050
Tool Palette 1051
Translation Manager 1052
Multi-line Editor 1054
Type Library Editor 1054
View Form 1057
View Units 1057
Window List 1057
New Edit Window 1057
Toggle Form/Unit 1058
Model View Window 1058
CodeGuard Log 1059
Desktops 1059
Dock Edit Window 1060
Find Reference Results 1060

Help Insight 1060

Show Borders 1060

Show Grid 1061

Show Tag Glyphs 1061

Snap To Grid 1061

Toolbars 1061

Translation Editor 1062

Welcome Page 1062

Win View 1062

Assembly Metadata Explorer (Reflection viewer) 1063

Type Library Explorer 1065

Search 1066

Keyboard Mappings 1068

Key Mappings 1068

BRIEF Keyboard Shortcuts 1069

IDE Classic Keyboard Shortcuts 1070

Default Keyboard Shortcuts 1073

Epsilon Keyboard Shortcuts 1076

Visual Basic Keyboard Shortcuts 1078

Visual Studio Keyboard Shortcuts 1079

Command Line Switches and File Extensions 1082

IDE Command Line Switches and Options 1082

File Extensions of Files Generated by RAD Studio 1084

Together Reference 1086

Together Configuration Options 1086

Configuration Levels 1088

Together Option Categories 1088

Option Value Editors 1101

Together Sequence Diagram Roundtrip Options 1102

Together Source Code Options 1103

System macros 1103

Together Keyboard Shortcuts 1104

GUI Components for Modeling 1105

Diagram View 1106

Pattern GUI Components 1107

Menus 1110

Quality Assurance GUI Components 1110

Model View 1112

Object Inspector 1113
Tool Palette 1114
Together Refactoring Operations 1115
Project Types and Formats with Support for Modeling 1116
UML 1.5 Reference 1116
UML 1.5 Activity Diagrams 1117
UML 1.5 Class Diagrams 1121
UML 1.5 Component Diagrams 1128
UML 1.5 Deployment Diagrams 1129
UML 1.5 Interaction Diagrams 1131
UML 1.5 Statechart Diagrams 1135
UML 1.5 Use Case Diagrams 1137
Together Glossary 1139
UML 2.0 Reference 1140
UML 2.0 Activity Diagrams 1140
UML 2.0 Class Diagrams 1143
UML 2.0 Component Diagrams 1145
UML 2.0 Composite Structure Diagrams 1146
UML 2.0 Deployment Diagrams 1148
UML 2.0 Interaction Diagrams 1149
UML 2.0 State Machine Diagrams 1155
UML 2.0 Use Case Diagrams 1157
Together Wizards 1158
Create Pattern Wizard 1159
New Together Project Wizards 1159
Pattern Wizard 1162
Index **a**

1 Concepts

The application development life cycle involves designing, developing, testing, debugging, and deploying applications. RAD Studio provides powerful tools to support this iterative process, including form design tools, the Delphi for compiler, an integrated debugging environment, and installation and deployment tools.

Topics

Name	Description
Compiling, Building, and Running Applications (↗ see page 2)	This section describes essential information about compiling, building, and running applications. As of 2007, the new build engine in the IDE is MSBuild.exe. You can now create and manage build configurations, named option sets, and targets files that enhance your ability to control your development environment.
Debugging Applications (↗ see page 10)	Many of the same techniques are used for debugging applications in different environments. RAD Studio provides an integrated debugging environment that enables you to debug Win32 application and .NET applications. In addition, you can use the debugger to debug an application running on a remote machine that does not have RAD Studio installed.
General Concepts (↗ see page 14)	This section provides an overview of basic concepts.
Getting Started (↗ see page 22)	The RAD Studio integrated development environment (IDE) provides many tools and features to help you build powerful applications quickly. Not all features and tools are available in all editions of RAD Studio. For a list of features and tools included in your edition, refer to the feature matrix on http://www.codegear.com.
Refactoring Applications (↗ see page 55)	Refactoring is a technique you can use to restructure and modify your code in such a way that the intended behavior of your code stays the same. RAD Studio provides a number of refactoring features that allow you to streamline, simplify, and improve both performance and readability of your application code.
Testing Applications (↗ see page 70)	Unit testing is an integral part of developing reliable applications. The following topics discuss unit testing features included in RAD Studio.
Modeling Applications with Together (↗ see page 81)	This section provides an overview of the features provided by Together. **Note:** The product version you have determines which Together features are available.

1.1 Compiling, Building, and Running Applications

This section describes essential information about compiling, building, and running applications. As of 2007, the new build engine in the IDE is MSBuild.exe. You can now create and manage build configurations, named option sets, and targets files that enhance your ability to control your development environment.

Topics

Name	Description
Compiling, Building, and Running Applications (↗ see page 2)	As you develop your application in the IDE, you can compile (or make), build, and run the application in the IDE. All three operations can produce an executable (such as `.exe`, `.dll`, `.obj`, or `.bpl`). However, the three operations differ slightly in behavior: • **Compile** (**Project**▶**Compile**) or, for C++, **Make** (**Project**▶**Make**) compiles only those files that have changed since the last build as well as any files that depend on them. Compiling or making does not execute the application (see **Run**). • **Build** (**Project**▶**Build**) compiles all of the... more (↗ see page 2)
MSBuild Overview (↗ see page 4)	To build projects, the IDE now uses MSBuild instead of the previous internal build system. The build, compile, and make commands available in the IDE call the new build engine from Microsoft: MSBuild, which provides thorough dependency checking. MSBuild project files are XML-based, and contain sections that describe specific Items, Properties, Tasks, and Targets for the project. For more information about MSBuild, see the Microsoft documentation at http://msdn.microsoft.com.
Build Configurations Overview (Delphi) (↗ see page 5)	Build configurations consist of options that you can set on all the build-related pages of the **Project**▶**Options** dialog box. Build configuration information is saved in the MSBuild project file (`.dproj` or `.groupproj`).
Build Configurations Overview (C++) (↗ see page 6)	Build configurations consist of options that you can set on all the build-related pages of the **Project**▶**Options** dialog box. Build configurations are saved in the project file (`.cbproj` or `.groupproj`).
Named Option Sets Overview (↗ see page 7)	Named option sets consist of options that you can create from and apply to the build-related pages of the **Project**▶**Options** dialog box. Named option sets are saved in files with the extension `.optset`.
Targets files (↗ see page 8)	A .targets file is an MSBuild-compliant XML file you can add to a project to allow customizing the build process. A .targets file can have **<Target>** nodes containing MSBuild scripts. You can also add or modify project property values with a .targets file. You can leverage the large variety of MSBuild tasks available in the .NET Framework SDK and on the internet, such as "Zip", "SVNCheckout", and "Mail", or write custom tasks yourself. In summary: • A .targets file is an XML file with **<Target>** nodes that contain MSBuild scripts with lists of tasks to perform. • You can create, add, enable,... more (↗ see page 8)

1.1.1 Compiling, Building, and Running Applications

As you develop your application in the IDE, you can compile (or make), build, and run the application in the IDE. All three operations can produce an executable (such as `.exe`, `.dll`, `.obj`, or `.bpl`). However, the three operations differ slightly in behavior:

- **Compile** (**Project▶Compile**) or, for C++, **Make** (**Project▶Make**) compiles only those files that have changed since the last build as well as any files that depend on them. Compiling or making does not execute the application (see **Run**).
- **Build** (**Project▶Build**) compiles all of the source code in the current project, regardless of whether any source code has changed. Building is useful when you are unsure which files have changed, or if you have changed project or compiler options.
- **Run** (**Run▶Run (F9)** compiles any changed source code and, if the compile is successful, executes your application, allowing you to use and test it in the IDE .

To delete all the generated files from a previous build, use the Clean command, available on the context menu of the project node in the **Project Manager**.

Compiler Options

You can set many of the compiler options for a project by choosing **Project▶Options** and selecting a compiler-related page. For example, most of the options on both the Delphi **Delphi Compiler** page and the **C++ Compiler** page correspond to compiler options that are described in the online Help for that page.

To specify additional compiler options, you can invoke the compiler from the command line. For a complete list of the Delphi compiler options and information about running the Delphi compiler from the command line, see **Delphi Language Guide** in the **Contents** pane.

Compiler Status and Information

To display the current compiler options in the **Messages** window each time you compile your project, choose **Tools▶Options▶Environment Options** and select **Show command line**. The next time you compile a project, the **Messages** window displays the command used to compile the project and the response file. The response file lists the compiler options and the files to be compiled.

After you compile a project, you can display information about it by choosing **Project▶Information**. The resulting **Information** dialog box displays the number of lines of source code compiled, the byte size of your code and data, the stack and file sizes, and the compile status of the project.

Compiler Errors

As you compile a project, compiler messages are displayed in the **Messages** window. For an explanation of a message, select the message and press `F1`.

Build Configurations

You can save certain project options as a named build configuration. You can specify the active build configuration for each of your projects, and you can change the active build configuration during project development. For example, you can set project options specific to debugging your project, save the options as the **Debug** build configuration, and then apply them as the active configuration while you debug the project.

The options in a build configuration are available on the **Compiler**, **Compiler Messages**, **Linker**, and **Conditionals/Directories** pages of the **Project▶Options** dialog box.

By default, the IDE includes a **Debug** configuration and a **Release** configuration. You can modify the options in both of these default configurations, and you can create new build configurations of your own. To apply a selected build configuration to specific projects or project groups, use the **Build Configuration Manager**.

Using MSBuild to Build Your Project

When you explicitly build a project, the IDE calls MSBuild, the Microsoft build engine. The build process is entirely transparent to developers. MSBuild is called as part of the Compile,Build, and Run commands available on the Project and Run menus. However, you can also invoke MSBuild.exe from the command line or using the MSBuild Console available on the **Start** menu.

See Also

MSBuild Overview (see page 4)

Build Configurations Overview (Delphi) (see page 5)

Build Configurations Overview (C++) (see page 6)

Building Packages (see page 105)

Build Configuration Manager (see page 828)

Creating Named Build Configurations (C++) (see page 107)

Creating Named Build Configurations (Delphi) (see page 108)

Building a Project from the Command Line (see page 108)

1.1.2 MSBuild Overview

To build projects, the IDE now uses MSBuild instead of the previous internal build system. The build, compile, and make commands available in the IDE call the new build engine from Microsoft: MSBuild, which provides thorough dependency checking. MSBuild project files are XML-based, and contain sections that describe specific Items, Properties, Tasks, and Targets for the project.

For more information about MSBuild, see the Microsoft documentation at http://msdn.microsoft.com.

Migrating Projects to MSBuild

If you open a pre-existing Delphi project (such as one with`.bdsproj` extension), the IDE automatically converts the project to use MSBuild and changes the project extension to `.dproj` for a Delphi project or to `.cbproj` for a C++ project.

Project groups are also converted to MSBuild and given the project-group extension `.groupproj`.

Building Projects

You can build projects without knowing anything about MSBuild because the IDE handles all the details for you. The **Project▶Compile** and **Project▶Build** commands both invoke MSBuild, but the scope of each command is different.

You can also explicitly build projects from the command line by running MSBuild.exe with your `.dproj` file.

To invoke MSBuild in a custom command environment, choose **Start menu▶RAD Studio Command Prompt**. This command window automatically sets both the path to the executable and the variable for your installation directory.

If you want to use MSBuild from the command line outside of the RAD Studio Command Prompt, you should set the following environment variables yourself:

```
BDS=c:\program  files\CodeGear\RAD  Studio\5.0  FrameworkDir=c:\Windows\Microsoft.NET\Framework
FrameworkVersion=v2.0.50727
```

Applying Your Own Custom Build Configurations

Several pages of the **Project▶Options** dialog box allow you to save groups of options into a named build configuration. Two default build configurations are **Debug** and **Release**. C++Builder supports a **Base** configuration as well. You can use the Configuration Manager to selectively apply any named build configuration as the active build configuration for your project or project group.

Setting Build Events and Viewing Build Output

When you build a project, the results of the build appear in the **Messages** window, on the **Output** tab. You can specify pre-build and post-build events using the **Project▶Options▶Build Events** dialog box (C++Builder supports pre-link events as well). If you specify build events, the commands you specified and their results also appear in the **Messages** window. To control the level of output from MSBuild, use the **Verbosity** field on the **Tools▶Options▶Environment Options** page.

File Type Determines Build Order

MSBuild builds a project using the following order:

1. .RC files
2. .ASM files
3. .PAS files
4. .CPP files

The build proceeds downward through the directory or folder nodes in the Project Manager. Within each folder, files are built in order according to their file type. You can control build order by placing files within different folders or within virtual folders in the Project Manager.

See Also

Compiling (see page 2)

Setting Project Options (C++) (see page 163)

Build Events Dialog Box (see page 829)

Running MSBuild on the Command Line (see page 108)

Overview of Build Configurations (Delphi) (see page 5)

Overview of Build Configurations (C++) (see page 6)

Creating Named Build Configurations (C++) (see page 107)

Applying the Active Build Configuration (see page 104)

Virtual Folders Overview (see page 50)

1.1.3 Build Configurations Overview (Delphi)

Build configurations consist of options that you can set on all the build-related pages of the **Project▶Options** dialog box. Build configuration information is saved in the MSBuild project file (`.dproj` or `.groupproj`).

IDE Contains Three Default Build Configurations

Base, **Debug**, and **Release** are the three default build configurations. **Base** acts as a base set of option values that is used in all the configurations you subsequently create. The **Debug** configuration enables optimization and debugging, as well as setting specific syntax options. The **Release** configuration doesn't produce symbolic debugging information as well as the code is not generated for TRACE and ASSERT calls, meaning the size of your executable is reduced.

You can change option values in any configuration, including **Base**. You can delete the **Debug** and **Release** configurations, but you cannot delete the **Base** configuration.

You can change the options for the three default configurations, and you can create and name your own custom build configurations on numerous build-related pages of the **Project▶Options** dialog box.

You Can Specify the Active Configuration for Your Projects

Every project in RAD Studio has an active build configuration associated with it, as well as any number of other inactive build configurations that you have created.

The active build configuration is used in Compile and Build commands for the project. Use the **Build Configuration Manager** to specify the active configuration for a selected project or project group (choose **Project**▶**Configuration Manager**).

See Also

MSBuild Overview (see page 4)

Creating Named Build Configurations (Delphi) (see page 108)

Applying the Active Build Configuration (see page 104)

1.1.4 Build Configurations Overview (C++)

Build configurations consist of options that you can set on all the build-related pages of the **Project**▶**Options** dialog box. Build configurations are saved in the project file (`.cbproj` or `.groupproj`).

Each Project Contains Default Build Configurations

Base, **Debug**, and **Release** are the three default build configurations. **Base** acts as a base set of option values that is used in all the configurations you subsequently create. The **Debug** configuration enables optimization and debugging, as well as setting specific syntax options. The **Release** configuration doesn't produce symbolic debugging information as well as the code is not generated for TRACE and ASSERT calls, meaning the size of your executable is reduced.

You can change option values in any configuration, including **Base**. You can delete the **Debug** and **Release** configurations, but you cannot delete the **Base** configuration.

You can change option values in any configuration, including **Base**. You can delete the **Debug** and **Release** configurations, but you cannot delete the **Base** configuration.

You Can Specify the Active Configuration for Your Projects

Every project has an active build configuration associated with it, as well as any number of other inactive build configurations that you have created.

The active build configuration is used in Compile and Build commands for the project. Use the **Build Configuration Manager** to specify the active configuration for a selected project or project group (choose **Project**▶**Configuration Manager**).

Build Configurations

Each configuration, except **Base**, is based on another configuration from which it inherits its values. The **Debug** and **Release** configurations inherit their values from **Base**.

You can, in turn, create a new configuration based on any configuration, and the new configuration inherits values from its parent. After creating a configuration, you can change its values to whatever you want, and you can make it the active configuration for a project or projects. You can also delete any configuration except **Base**.

Unless it is changed, a configuration inherits its option values from its parent configuration. This inheritance is not static: if the parent configuration changes, so do its children for all inherited values.

The default value of an option is its value in its parent configuration. You can revert an option to its default value.

Comparing Configurations and Option Sets

You can also save a configuration's option values to a file as a *named option set* using a file save dialog. You can apply option set values to any configuration in any project.

Note that a configuration is different from an option set, though they are related. Both consist of sets of option values. The main distinction is that configurations are associated with projects, whereas option sets are saved in files independent of projects. Build configuration values are stored in the project file, so saving a project saves changes to configurations, but option sets are unaffected. Changing a project's configurations and adding or deleting configurations does not affect option sets. Similarly, saving option sets does not change configurations.

The **Build Configuration** drop-down menu lists configurations for that project—not option sets. Each project has its own list of configurations, independent of other projects. However, you can access any option set from any project.

Configuration and Option Set Values

Note that configurations and option sets may not contain values for all possible project options--they contain *only* the options that are different from the parent configuration. **Base** also does not contain values for all possible options.

If an option value is not in a configuration, the IDE looks in its parent configuration, then the parent's parent configuration, and so on. If not found in any of the configurations in the inheritance chain, the value comes from the appropriate tool that is being configured.

For instance, if a configuration inheritance chain does not include a value for a particular compiler option, the default value is specified by the compiler itself. When you save a configuration or option set, only its values are saved--not values for every option.

You Can Merge Some Options from Parent Configuration

Some options that contain a list of items, such as defines or paths, have a **Merge** check box.

If **Merge** is checked, the IDE merges the option's list with that of its immediate ancestor's configuration's list for that option. Note that the IDE does not actually change the contents of the option, but acts as if the list included the ancestor's list. If the ancestor's **Merge** check box is also checked, the IDE merges this ancestor's list for that option, and so on up the inheritance chain.

If **Merge** is unchecked, the IDE uses only the items in the option.

Some Options are No Longer Available

Some options available in previous releases are no longer available, except possibly through the tool option flags. See Unavailable Options (↗ see page 893) for more information.

See Also

MSBuild Overview (↗ see page 4)

Creating Named Build Configurations (C++) (↗ see page 107)

Applying the Active Build Configuration (↗ see page 104)

Working With Named Option Sets (↗ see page 113)

Setting C++ Project Options (↗ see page 163)

Unavailable Options (↗ see page 893)

1.1.5 Named Option Sets Overview

Named option sets consist of options that you can create from and apply to the build-related pages of the **Project▶Options** dialog box. Named option sets are saved in files with the extension `.optset`.

Creating Option Sets

You can save a project's option values to a file as a *named option set* using the context menu in the **Project Manager** and the Save button in **Project Option** dialogs.

Comparing Option Sets and Configurations

Option sets are different from configurations, though they are related. Both consist of sets of option values. The main distinction is that configurations are part of the project file; option sets are saved in files independent of projects. Saving a project saves changes to configurations, but option sets are unaffected. Changing a project's configurations and adding or deleting configurations does not affect option sets. Similarly, saving option sets does not change configurations. Each project has its own list of configurations--not option sets--independent of other projects. You can access any option set from any project.

Configuration and Option Set Values

Configurations and option sets may not contain values for all possible project options. When you save a configuration or option set, only the values that are different from the parent configuration are saved--not values for every option. An option set file does not refer to any project's configuration, since option sets are independent of configurations.

For instance, suppose you had a project open in the IDE and the active configuration was the **Base** configuration. You change one option and then save the option set. The only option values saved in the option set file are the one option value you changed plus the option values in the **Base** configuration that are different from the defaults.

Applying Option Sets

You can apply option set values to any configuration in any project using the **Apply Option Set** dialog. You have three choices for how you apply values: **Overwrite**, **Replace**, or **Preserve**.

- **Overwrite** replaces the current configuration with the values from the option set entirely; values not in the option set get their default values.
- **Replace** writes all the values from the option set to the current configuration, but no other values are changed.
- **Preserve** writes only the values from the option set that are not already set in the active configuration.

See Also

Working With Named Option Sets (see page 113)

Apply Option Set dialog (see page 828)

Setting C++ Project Options (see page 163)

Creating Named Build Configurations (see page 107)

Applying the Active Build Configuration (see page 104)

1.1.6 Targets files

A .targets file is an MSBuild-compliant XML file you can add to a project to allow customizing the build process. A .targets file can have **<Target>** nodes containing MSBuild scripts.

You can also add or modify project property values with a .targets file. You can leverage the large variety of MSBuild tasks available in the .NET Framework SDK and on the internet, such as "Zip", "SVNCheckout", and "Mail", or write custom tasks yourself.

In summary:

- A .targets file is an XML file with **<Target>** nodes that contain MSBuild scripts with lists of tasks to perform.
- You can create, add, enable, or remove .targets files in a project with the IDE.

Adding .targets files

You create and add a .targets file to the project using either menu commands or the **Project Manager** context menu. The IDE generates a minimal .targets file containing only the **<Project>** root node and namespace attribute. You can then add MSBuild scripts inside the **<Project>** node.

By default, .targets files are added to the project but not used by it. You enable a .targets file with the **Project Manager**, which adds the .targets file as an MSBuild **<Import>** to your project. All .targets files must contain valid MSBuild scripts free of errors. If the file has any errors, you are notified and, if the project references the invalid .targets file, it is disabled and cannot be re-enabled until the errors are corrected. Because MSBuild can only read **<Import>**s directly from disk, the .targets file must be saved to disk before a Make, Build, or invoking any of its targets.

Target element in .targets files

The target element in the .targets file contains a set of tasks for MSBuild to execute. Here is its format:

```
<Target Name="Target Name"
        DependsOnTargets="DependentTarget"
        Inputs="Inputs"
        Outputs="Outputs"
        Condition="'String A' == 'String B'">
    <Task>... </Task>
    <OnError... />
</Target>
```

See http://msdn2.microsoft.com/en-us/library/t50z2hka.aspx for more information on target elements.

Using .targets files

When a .targets file contains valid **<Target>** elements, you can invoke those targets using MSBuild from the **Project Manager**, provided the .targets file is enabled.

The .targets file can declare its own properties, targets, and item groups for use by its targets and tasks. It can also refer to properties and item groups in the project file, including those imported from the standard Codegear .targets file in the Windows directory at **microsoft.net\Framework\v2.0.xxx\Borland.Cpp.Targets**. You should not modify any .targets files in this directory, since improper edits may cause the IDE to stop functioning correctly.

Tip: For more information about .targets files, see http://msdn2.microsoft.com/en-us/library/ms164312.aspx.

See Also

Using Targets Files (see page 109)

1.2 Debugging Applications

Many of the same techniques are used for debugging applications in different environments. RAD Studio provides an integrated debugging environment that enables you to debug Win32 application and .NET applications. In addition, you can use the debugger to debug an application running on a remote machine that does not have RAD Studio installed.

Topics

Name	Description
Overview of Debugging (↗ see page 10)	RAD Studio includes both the CodeGear .NET Debugger and CodeGear Win32 Debugger. The IDE automatically uses the appropriate debugger based on the active project type. Cross-platform debugging within a project group is supported and, where possible, the debuggers share a common user interface. The integrated debuggers let you find and fix both runtime errors and logic errors in your RAD Studio application. Using the debuggers, you can step through code, set breakpoints and watches, and inspect and modify program values. As you debug your application, the debug windows are available to help you manage the debug session and provide information... more (↗ see page 10)
Overview of Remote Debugging (↗ see page 12)	Remote debugging enables you to debug one or more applications on a remote machine when the IDE is running only on your local machine. This allows debugging on a machine where it is impractical to install the entire IDE and rebuild a project. Remote debugging is useful for applications that run differently on your local machine than on an end user's machine.

1.2.1 Overview of Debugging

RAD Studio includes both the CodeGear .NET Debugger and CodeGear Win32 Debugger. The IDE automatically uses the appropriate debugger based on the active project type. Cross-platform debugging within a project group is supported and, where possible, the debuggers share a common user interface.

The integrated debuggers let you find and fix both runtime errors and logic errors in your RAD Studio application. Using the debuggers, you can step through code, set breakpoints and watches, and inspect and modify program values. As you debug your application, the debug windows are available to help you manage the debug session and provide information about the state of your application.

The **Debug Inspector** enables you to examine various data types such as arrays, classes, constants, functions, pointers, scalar variables, and interfaces. To use the **Debug Inspector**, select **Run▶Inspect**.

Stepping Through Code

Stepping through code lets you run your program one line of code at a time. After each step, you can examine the state of the program, view the program output, modify program data values, and continue executing the next line of code. The next line of code does not execute until you tell the debugger to continue.

The Run menu provides the Trace Into and Step Over commands. Both commands tell the debugger to execute the next line of code. However, if the line contains a function call, Trace Into executes the function and stops at the first line of code *inside* the function. Step Over executes the function, then stops at the first line *after* the function.

Evaluate/Modify

The Evaluate/Modify functionality allows you to evaluate an expression. You can also modify a value for a variable and insert that value into the variable. The Evaluate/Modify functionality is customized for the language you are using:

- For a C++ project, the Evaluate/Modify dialog accepts only C++ expressions.

- For a C# project, the Evaluate/Modify dialog accepts only C# expressions.
- For a Delphi project, the Evaluate/Modify dialog accepts only Delphi expressions.

Breakpoints

Breakpoints pause program execution at a certain point in the program or when a particular condition occurs. You can then use the debugger to view the state of your program, or step over or trace into your code one line or machine instruction at a time. The debugger supports four types of breakpoints:

- Source breakpoints pause execution at a specified location in your source code.
- Address breakpoints pause execution at a specified memory address.
- Data breakpoints allow you to pause execution when memory at a particular address changes.
- Module load breakpoints pause execution when the specified module loads.

Note: Data breakpoints are available only for the Win32 debugger.

Watches

Watches lets you track the values of program variables or expressions as you step over or trace into your code. As you step through your program, the value of the watch expression changes if your program updates any of the variables contained in the watch expression.

Debug Windows

The following debug windows are available to help you debug your program. By default, most of the windows are displayed automatically when you start a debugging session. You can also view the windows individually by selecting **View ▶ Debug Windows**.

Each window provides one or more right-click context menus. The `F1` Help for each window provides detailed information about the window and the context menus.

Debug Window	Description
Breakpoint List	Displays all of the breakpoints currently set in the **Code Editor** or **CPU** window.
Call Stack	Displays the current sequence of function calls.
Watch List	Displays the current value of watch expressions based on the scope of the execution point.
Local Variables	Displays the current function's local variables, enabling you to monitor how your program updates the values of variables as the program runs.
Modules	Displays processes under control of the debugger and the modules currently loaded by each process. It also provides a hierarchical view of the namespaces, classes, and methods used in the application.
Threads Status	Displays the status of all processes and threads of execution that are executing in each application being debugged. This is helpful when debugging multi-threaded applications.
Event Log	Displays messages that pertain to process control, breakpoints, output, threads, and module.
CPU	Displays the low-level state of your program, including the assembly instructions for each line of source code and the contents of certain registers.
FPU	Displays the contents of the Floating-point Unit and SSE registers in the CPU.

Remote Debugging

Remote debugging lets you debug an application running on a remote computer. Your computer must be connected to the remote computer through TCP/IP and the remote debugger must be installed on the remote machine. After you create and copy the required application files to the remote computer, you can connect to that computer and begin debugging.

See Also

Overview of Remote Debugging (↗ see page 12)

Preparing a Project for Debugging (↗ see page 124)

Setting and Modifying Breakpoints (↗ see page 118)

Adding a Watch (↗ see page 116)

Modifying Variable Expressions (↗ see page 124)

Inspecting and Changing the Value of Data Elements (↗ see page 122)

Debugging Remote Applications (↗ see page 125)

Attaching to a Running Process (↗ see page 117)

1.2.2 Overview of Remote Debugging

Remote debugging enables you to debug one or more applications on a remote machine when the IDE is running only on your local machine. This allows debugging on a machine where it is impractical to install the entire IDE and rebuild a project. Remote debugging is useful for applications that run differently on your local machine than on an end user's machine.

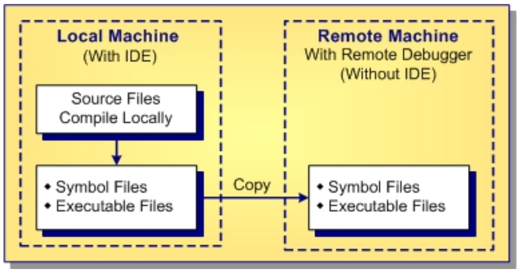
The Remote Debugger Executable

The remote debugger executable is named rmtdbg105.exe. The executable and its supporting files must be present on the remote machine. The easiest way to install the executable is directly from the RAD Studio installation disk. However, if the installation disk is not available, you can copy the required files from a machine that has the full RAD Studio IDE installed.

Local and Remote Files

Three types of files are involved in remote debugging:

- Source files
- Executable files
- Symbol files



Source files are compiled using the IDE on the local machine. The executable files and symbol files produced after compilation must be copied to the remote machine.

Source Files

When you debug a project on a remote machine, the source files for the project must be open on the local machine. The source files display in the editor window to show a program's current execution point. You do not use source files on the remote machine.

Executable Files

Executable files are the .dll files and .exe files that are mapped into the application's address space. You generate these files on the local machine, then copy them to the remote machine.

Symbol Files

Symbol files are generated on the local machine at compile time. These are used by the debugger to get information such as the mapping of machine instructions to source line numbers or the names and types of variables declared in the source files. The extension for the symbol files depends on the language, as shown in the following table:

Language	Debug symbol file extension
Delphi for Win32	.rsm
Delphi for .NET	.rsm and .pdb
C++	.tds
C#	.pdb

You must set up specific options to generate symbol files on the local machine, then copy the files to the remote machine.

Local and Remote Machines

To use remote debugging, you must be able to log on to the remote machine and you must have write access to at least one directory.

Note: The remote debugger does not provide a mechanism for interacting with an application on the remote machine. If you need to interact with the application, you must establish a remote desktop connection.

See Also

Debugging a Remote Application (↗ see page 125)

Installing a Debugger on a Remote Machine (↗ see page 126)

Establishing a Connection for Remote Debugging (↗ see page 127)

Preparing Files for Remote Debugging (↗ see page 128)

1.3 General Concepts

This section provides an overview of basic concepts.

Topics

Name	Description
Managing the Development Cycle Overview (↗ see page 14)	The development cycle as described here is a subset of Application Lifecycle Management (ALM), dealing specifically with the part of the cycle that includes the implementation and control of actual development tasks. The development cycle described here does not include such things as modeling applications. The tools of ALM include: • Requirements management • Source control • User interface design • Code visualization capabilities • Project building, compilation, and debugging capabilities
Designing User Interfaces (↗ see page 15)	A graphical user interface (GUI) consists of one or more windows that let users interact with your application. At designtime, those windows are called *forms*. RAD Studio provides a designer for creating Windows Forms, Web Forms, VCL Forms, and HTML pages. The Designer and forms help you create professional-looking user interfaces quickly and easily.
Using Source Control (↗ see page 16)	This topic provides an overview of general source control concepts that are consistent among a number of source control systems, also known as automated change and software configuration management (SCM) systems. .
Localizing Applications (↗ see page 18)	RAD Studio includes a suite of Translation Tools to facilitate localization and development of .NET and Win32 applications for different locales. The Translation Tools include the following: • Satellite Assembly Wizard (for .NET) • Resource DLL Wizard (for Win32) • Translation Manager • Translation Repository The Translation Tools are available for Delphi VCL Forms applications (both .NET and Win32), and Win32 console applications, packages, and DLLs. You can access the Translation Tools configuration options by choosing **Tools ▶ Options ▶ Translation Tools Options**.
Deploying Applications (↗ see page 19)	After you have written, tested, and debugged your application, you can make it available to others by deploying it. Depending on the size and complexity of the application, you can package it as one or more assemblies, as compressed cabinet (`.cab`) files, or in an installer program format (such as `.msi`). After the application is packaged, you can distribute it by using XCOPY, FTP, as a download, or with an installer program. For additional information about deploying specific types of applications, refer to the list of links at the end of this topic.

1.3.1 Managing the Development Cycle Overview

The development cycle as described here is a subset of Application Lifecycle Management (ALM), dealing specifically with the part of the cycle that includes the implementation and control of actual development tasks. The development cycle described here does not include such things as modeling applications.

The tools of ALM include:

- Requirements management
- Source control
- User interface design
- Code visualization capabilities
- Project building, compilation, and debugging capabilities

Requirements Management

Requirements management tools enable you to add, remove, and update requirements for your software project. A fully integrated tool also enables you to create links between the requirement specification and the portions of the code within your software project that fulfill the requirement.

Source Control

A source control system allows you to manage versions or renditions of your project files. Most source control systems maintain a central repository of code and allow you to check-in, check-out, update, commit, and otherwise manage your source files.

User Interface Design

RAD Studio provides a rich environment for designing a .NET user interface. In addition to the Windows Form Designer, which includes a full set of visual components, the IDE gives you tools to build ASP.NET Web Forms, along with a set of Web Controls.

The IDE also includes a VCL.NET Forms design tool, which allows you to build .NET applications using VCL components. The Designer offers a variety of alignment tools, font tools, and visual components for building many types of applications, including MDI and SDI applications, tabbed dialogs, and data aware applications.

Code Visualization

The Code Visualization feature of RAD Studio provides the means to document and debug your class designs using a visual paradigm. As you load your projects and code files, you can use the **Model View** to get both a hierarchical graphical view of all of the objects represented in your classes, as well as a UML-like model of your application objects. This feature can help you visualize the relationships between objects in your application, and can assist you in developing and implementing.

Compile, Build, Run, and Debug

RAD Studio provides MSBuild, the industry standard build engine, along with an integrated debugger. You can build only the changed elements of the project by using the Compile command. To build the entire project regardless of changes, use the Build command. Projects with subprojects and multiple source files can be built all together, or you can build each project individually.

The integrated debugger allows you to set watches and breakpoints, and to step through, into, and over individual lines of code. A set of debugger windows provides details on variables, processes, and threads, and lets you drill down deeply into your code to find and fix errors.

See Also

Compiling (↗ see page 2)

Building Packages (↗ see page 105)

Debugging Applications (↗ see page 10)

1.3.2 Designing User Interfaces

A graphical user interface (GUI) consists of one or more windows that let users interact with your application. At designtime, those windows are called *forms*. RAD Studio provides a designer for creating Windows Forms, Web Forms, VCL Forms, and HTML pages. The Designer and forms help you create professional-looking user interfaces quickly and easily.

Using the Designer

When you create a Windows, Web, or Web Services application, the IDE automatically displays the appropriate type of form on the **Design** tab in the IDE. As you drop components, such as labels and text boxes, from the **Tool Palette** on to the form, RAD Studio generates the underlying code to support the application. You can use the **Object Inspector** to modify the properties of components and the form. The results of those changes appear automatically in the source code on the **Code** tab. Conversely, as you modify code with **Code Editor**, the changes you make are immediately reflected on the **Design** tab.

The **Tool Palette** provides dozens of controls to simplify the creation of Windows Forms, Web Forms, and HTML pages. When creating a Windows Form, for example, you can use the MainMenu component to create a customized main menu in minutes. After placing the component on a Windows Form, you type the main menu entries and commands in the boxes provided. The ContextMenu component provides similar functionality for creating context menus. There are also several dialog box components for commonly performed functions, such as opening and saving files, setting fonts, selecting colors, and printing. Using these components saves time and provides a consistent look and feel for the dialogs in your application.

As you design your user interface, you can undo and repeat previous changes to a form by choosing **Edit▶Undo** and **Edit▶Redo**. When you are satisfied with the appearance of the form, you can lock the components and form to prevent accidental changes by right-clicking the form and choosing Lock Controls.

Setting Designer Options

You can set options that effect the appearance and behavior of the Designer. For example, you can adjust the grid settings, and the style of generated code and HTML. To set these options, choose **Tools▶Options** and then use the **Windows Form Designer** and **HTML Option** dialog boxes.

Setting Designer Guidelines with VCL Components

You can use VCL or VCL.NET (with Delphi or C++) to setup components that are "aware" of their relation to other components on a form. For instance, when you drop a component on a form, it will leave a certain amount of space from the border of the form, depending on how the 'padding' property is set.

You can set properties to specify the distance between controls, shortcuts, focus labels, tab order, and maximum number of items (listboxes, menus).

The Code Developer can then use these components to create forms. when the Use Designer Guidelines option is enabled. If the Snap to Grid option is enabled, and Use Designer Guidelines is also enabled, the designer guidelines will take precedence.

See the Creating Designer Guidelines link at the end of this topic, to view the procedure for setting these guidelines.

See Also

ASP .NET Overview

Adding Components to a Designer (↗ see page 152)

Using Design Guidelines with VCL Components (↗ see page 165)

1.3.3 Using Source Control

This topic provides an overview of general source control concepts that are consistent among a number of source control systems, also known as automated change and software configuration management (SCM) systems. .

Source Control Basics

Each source control system consists of one or more centralized repositories and a number of clients. A repository is a database that contains not only the actual data files, but also the structure of each project you define.

Most source control systems adhere to a concept of a logical *project*, within which files are stored, usually in one or more tree directory structures. A source control system project might contain one or many RAD Studio projects in addition to other documents and artifacts. The system also enforces its own user authentication or, very often, takes advantage of the authentication provided by the underlying operating system. Doing so allows the source control system to maintain an audit trail or snapshot of updates to each file. These snapshots are typically referred to as *diffs*, for differences. By storing only the differences, the source control system can keep track of all changes with minimal storage requirements. When you want to see a complete copy of your file, the system performs a merge of the differences and presents you with a unified view. At the physical level, these differences are kept in separate files until you are ready to permanently merge your updates, at which time you can perform a *commit* action.

This approach allows you and other team members to work in parallel, simultaneously writing code for multiple shared projects, without the danger of an individual team member's code changes overwriting another's. Source control systems, in their most basic form, protect you from code conflicts and loss of early sources. Most source control systems give you the tools to manage code files with check-in and check-out capabilities, conflict reconciliation, and reporting capabilities. Most systems do not include logic conflict reconciliation or build management capabilities. For details about your particular source control system capabilities, refer to the appropriate product documentation provided by your source control system vendor.

Commonly, source control systems only allow you to compare and merge revisions for text-based files, such as source code files, HTML documents, and XML documents. Some source control systems allow you to include binary files, such as images or compiled code, in the projects you place under control. You cannot, however, compare or merge revisions of binary files. If you need to do more than store and retrieve specific revisions of of these types of files, you might consider creating a manual system for keeping tracking of the changes you make to binary files.

Repository Basics

Source control systems store copies of source files and difference files in some form of database repository. In some systems, such as CVS or VSS, the repository is a logical structure that consists of a set of flat files and control files. In other systems, the repositories are instances of a particular database management system (DBMS) such as InterBase, Microsoft Access, MS SQL Server, IBM DB2, or Oracle.

Repositories are typically stored on a remote server, which allows multiple users to connect, check files in and out, and perform other management tasks simultaneously. You need to make sure that you establish connectivity not only with the server, but also with the database instance. Check with your network, system, and database administrators to make sure your machine is equipped with the necessary drivers and connectivity software, in addition to the client-side source control software.

Some source control systems allow you to create a local repository in which you can maintain a snapshot of your projects. Over time the local image of your projects differs from the remote repository. You can establish a regular policy for merging and committing changes from your local repository to the remote repository.

Generally, it is not safe to give each member of your team a separate repository on a shared project. If you are each working on completely separate projects and you want to keep each project under source control locally, you can use individual local repositories. You can also create these multiple repositories on a remote server, which provides centralized support, backup, and maintenance.

Working with Projects

Source control systems, like development environments, use the project concept to organize and track groups of related files. No matter which source control system you use, you create a project that maintains your file definitions and locations. You also create projects in RAD Studio to organize the various assemblies and source code files for any given application. RAD Studio stores the project parameters in a project file. You can store this file in your source control system project, in addition to the various code files you create. You might share your project file among all the developers on your team, or you might each maintain a separate project file.

Most source control systems consider development environment project files to be binary, whether they are actually binary files or not. As a consequence, when you check a project file into a source control system repository, the source control system overwrites older versions of the file with the newer one without attempting to merge changes. The same is true when you pull a project, or check out the project file; the newer version of the project file overwrites the older version without merging.

Working with Files

The file is the lowest-level object that you can manage in a source control system. Any code you want to maintain under source control must be contained in a file. Most source control systems store files in a logical tree structure. Some systems, such as CVS, actually use terms like *branch*, to refer to a directory level. You can create files in a RAD Studio project and include them in your source control system, or you can pull existing files from the source control system. You can put an entire directory into the source control system, then you can check out individual files, multiple files, or entire subdirectory trees. RAD Studio gives you control over your files at two levels—at the project level within RAD Studio and in the source control system, through the RAD Studio interface to the source control system.

Note: The History View

provides revision information for your local source files. The **History View** can be used to track changes you make to files as you work on them in the Designer or the **Code Editor**.

See Also

Using the History Manager (↗ see page 149)

1.3.4 Localizing Applications

RAD Studio includes a suite of Translation Tools to facilitate localization and development of .NET and Win32 applications for different locales. The Translation Tools include the following:

- Satellite Assembly Wizard (for .NET)
- Resource DLL Wizard (for Win32)
- Translation Manager
- Translation Repository

The Translation Tools are available for Delphi VCL Forms applications (both .NET and Win32), and Win32 console applications, packages, and DLLs. You can access the Translation Tools configuration options by choosing **Tools ▶ Options ▶ Translation Tools Options**.

The Wizards

Before you can use the Translation Manager or Translation Repository, you must add languages to your project by running either the Satellite Assembly Wizard for .NET projects or the Resource DLL Wizard for Win32 projects. The Satellite Assembly Wizard creates a .NET satellite assembly for each language you add. The Resource DLL Wizard creates a Win32 resource DLL for each language. For simplicity, this documentation uses the term *resource module* to refer to either a satellite assembly or a resource DLL.

While running either wizard, you can include extra files, such as `.resx` or `.rc` files, that are not normally part of a project. You can add new resource modules to a project at any time. If you have multiple projects open in the IDE, you can process several at once.

You can also use the wizards to remove languages from a project and restoring languages to a project.

Translation Manager

After resource modules have been added to your project, you can use the Translation Manager to view and edit VCL forms and resource strings. After modifying your translations, you can update all of your application’s resource modules.

The External Translation Manager (ETM) is a version of the Translation Manager that you can set up and use without the IDE. ETM has the same functionality as the Translation Manager, with some additional menus and toolbars.

Translation Repository

The Translation Repository provides a database for translations that can be shared across projects, by different developers. While working in the Translation Manager, you can store translated strings in the Repository and retrieve translated strings from the Repository.

By default, each time your assemblies are updated, they will be populated with translations for any matching strings that exist in the Repository. You can also access the Repository directly, through its own interface, to find, edit, or delete strings.

The Translation Repository stores data in XML format. By default, the file is named `default.tmx` and is located in the RAD Studio`\bin` directory.

Files Generated by the Translation Tools

The files generated by the Translation Tools include the following:

File extension	Description
`.nfn` (.NET) `.dfn` (Win32)	The Translation Tools maintain a separate file for each form in your application and each target language. These files contain the data (including translated strings) that you see in the Translation Manager.
`.resx` (.NET)	The Satellite Assembly Wizard uses the compiler-generated `.drcil` file to create an `.resx` file for each target language. These `.resx` files contain special comments that are used by the Translation Tools.
`.rc` (Win32)	The Resource DLL Wizard uses the compiler-generated `.drc` file to create an `.resx` file for each target language. These `.resx` files contain special comments that are used by the Translation Tools.
`.tmx`	The Translation Repository stores data in an .tmx file. You can maintain more than one repository by saving multiple `.tmx` files.
`.bdsproj`	The External Translation Manager lists the assemblies (languages) and resources to be translated into a `.bdsproj` project file. When third-party translators add and remove languages from a project, they can save these changes in an `.bdsproj` file, which they return to the developer.

Note: You should not edit any of these files manually.

See Also

Adding Languages to a Project (see page 169)

Updating Resource Modules (see page 173)

Editing Resource Files in the Translation Manager (see page 170)

Setting the Active Language (see page 170)

Setting Up the External Translation Manager (see page 172)

1.3.5 Deploying Applications

After you have written, tested, and debugged your application, you can make it available to others by deploying it. Depending on the size and complexity of the application, you can package it as one or more assemblies, as compressed cabinet (`.cab`) files, or in an installer program format (such as `.msi`). After the application is packaged, you can distribute it by using XCOPY, FTP, as a download, or with an installer program.

For additional information about deploying specific types of applications, refer to the list of links at the end of this topic.

Deploying .NET Applications

Assuming that the target computer already has the .NET Framework installed on it, deploying a simple application that consists of a single executable is as easy as copying the `.exe` file to the target computer. You don't need to register the application and deleting the application files effectively uninstalls it.

Applications That Include Shared Assemblies

If your application includes an assembly that will be shared by other applications, you will need to uniquely identify the assembly with a strong name and then install it in the Global Assembly Cache (GAC). The strong name consists of the assembly's text name, version number, optional culture information, and the public key and digital signature to ensure uniqueness. The .NET Framework SDK provides command line utilities for creating a public/private key (`sn.exe`), assigning a strong name (`al.exe`), and installing an assembly in the GAC (`gacutil.exe`). For more information about these utilities, see the Framework SDK online Help.

Deploying VCL.NET Applications

When building applications that use the VCL .NET framework, the way you build the application determines what files you need to distribute with it. If you build the application by compiling VCL for .NET units directly into the program executable file, the application will have external dependencies only on the .NET Framework.

However, if you build the application by compiling the application to have external references to VCL for .NET assemblies, the application will have external dependencies on the .NET Framework, the `Borland.Delphi.dll`, and whatever RAD Studio packages you have added to the project references, for example, `Borland.VclRtl.dll` or `Borland.Vcl.dll`.

Deploying ASP.NET Applications

RAD Studio includes the ASP.NET Deployment Manager to assist you in deploying ASP.NET applications. You can use it to deploy to a remote computer by using a share or an FTP connection, or to your local computer. When you add a Deployment Manager to your project, an XML file (`.bdsdeploy)` is added to the project directory and a **Deploy** tab is added to the IDE. You provide destination and connection information on the **Deploy** tab and optionally modify the suggested list of files to copy, then the Deployment Manager copies the files to the deployment destination.

Redistributing the .NET Framework

If you plan to deploy your application to a computer that does not have the .NET Framework installed on it, you will need to redistribute and install the .NET Framework with your application. Microsoft provides a redistributable installer called `dotnetfx.exe`, which contains the common language runtime and .NET Framework components required to run .NET applications. For more information about `dotnetfx.exe`, see the .NET Framework SDK online Help.

Before Deploying a C# Application

Typically, while developing a C# application, you compile it with debugging information to facilitate testing. When you create a new project, it uses the default **Debug** option set, which creates the executable files and the program database file (`.pdb`) for debugging in the *project*\bin\Debug directory.

When you are ready to deploy the C# application, you can compile it using the default or a user-defined **Release** option set to create an optimized version of the application in the *project*\bin\Release directory. The optimized application is smaller, faster, and more efficient. To change the **Debug/Release** option sets, choose **Project**▶**Options**.

Deploying Win32 Applications

For information on deploying Win32 applications, refer to the **Deploying Win32 Applications** link at the end of this topic.

Using Installation Programs

For complex applications that consist of multiple files, you can use an installation program. Installation programs perform various tasks, such as copying executable and supporting files to the target computer and making Windows registry entries.

Setup toolkits, such as InstallAware, automate the process of creating installation programs, often without the need to write any code. InstallAware is based on Windows Installer (MSI) technology and can be installed from the RAD Studio installation DVD. After installing it, refer to the InstallAware online Help for information about using the product.

Redistributing RAD Studio Files

Many of the files associated with RAD Studio applications are subject to redistribution limitations or cannot be redistributed at all. Refer to the following documents for the legal stipulations regarding the redistribution of these files.

File	Description
deploy.htm	Contains deployment considerations for each edition of RAD Studio.
license.txt	Addresses legal rights and obligations concerning RAD Studio.
readme.htm	Contains last minute information about RAD Studio, possibly including information that could affect the redistribution rights for RAD Studio files.

These files are located, by default, at `C:\Program Files\CodeGear\RAD Studio\5.0`.

Redistributing Third Party Software

The redistribution rights for third party software, such as components, utilities, and helper applications, are governed by the vendor that supplies the software. Before you redistribute any third party software with your RAD Studio application, consult the third party vendor or software documentation for information regarding redistribution.

See Also

Deploying Database Applications for the .NET Framework

Deploying ASP.NET Applications for the .NET Framework

Deploying COM Interop Applications

Deploying Win32 Applications

1.4 Getting Started

The RAD Studio integrated development environment (IDE) provides many tools and features to help you build powerful applications quickly. Not all features and tools are available in all editions of RAD Studio. For a list of features and tools included in your edition, refer to the feature matrix on http://www.codegear.com.

Topics

Name	Description
What is RAD Studio? (see page 23)	RAD Studio is an integrated development environment (IDE) for building Delphi Win32 applications. The RAD Studio IDE provides a comprehensive set of tools that streamline and simplify the development life cycle. The tools available in the IDE depend on the version of RAD Studio you are using. The following sections briefly describe these tools.
What's New in RAD Studio (Delphi for Microsoft .NET) (see page 24)	RAD Studio provides key new features for developing Delphi for Microsoft .NET applications.
What's New in RAD Studio (C++Builder) (see page 28)	RAD Studio provides key new features for developing C++ applications.
What's New in RAD Studio (Delphi) (see page 32)	RAD Studio provides key new features for developing Delphi applications for Win32.
Tour of the IDE (see page 34)	When you start RAD Studio, the integrated development environment (IDE) launches and displays several tools and menus. The IDE helps you visually design user interfaces, set object properties, write code, and view and manage your application in various ways. The default IDE desktop layout includes some of the most commonly used tools. You can use the View menu to display or hide certain tools. You can also customize and save the desktop layouts that work best for you. The tools available in the IDE depend on the edition of RAD Studio you are using and include the following: • Welcome Page... more (see page 34)
IDE on Windows Vista (see page 40)	The IDE includes support for many new Vista user interface features including: • Vista-style Open File, Save File, and Task dialog boxes. • Vista theming. • AERO glass effects in controls.
Tools Overview (see page 41)	RAD Studio provides several developer tools that are available in the IDE and as standalone executables, as described in the following table.
Code Editor (see page 42)	The **Code Editor** occupies the center pane of the IDE window. The **Code Editor** is a full-featured, customizable, UTF8 editor that provides syntax highlighting, multiple undo capability, and context-sensitive Help for language elements. As you design the user interface for your application, RAD Studio generates the underlying code. When you modify object properties, your changes are automatically reflected in the source files. Because all of your programs share common characteristics, RAD Studio auto-generates code to get you started. You can think of the auto-generated code as an outline that you can examine to create your program. The **Code Editor** provides... more (see page 42)
Form Designer (see page 46)	The **Form Designer** or Designer, is displayed automatically when you are creating or editing a form. To access the Designer, click the **Design** tab at the bottom of the main editing window. The appearance and functionality of the Designer depends on the type of form you are creating or editing. For example, if you are using an HTML Element, you can display the HTML Tag Editor in the Designer by selecting **View ▶ Tag Editor**.

Starting a Project (↗ see page 47)	A project is a collection of files that is used to create a target application. This collection consists of the files you include and modify directly, such as source code files and resources, and other files that RAD Studio maintains to store project settings, such as the `.dproj` project file. Projects are created at design time, and they produce the project target files (.exe, .dll, .bpl, etc.) when you compile the project.To assist in the development process, the **Object Repository** offers many pre-designed templates, forms, files, and other items that you can use to create applications. To create a project,... more (↗ see page 47)
Template Libraries (↗ see page 50)	RAD Studio allows you to create multiple custom template libraries to use as the basis for creating future projects. Template libraries let you declare how a project can look, and enable you to add new types of projects to the **New Items** dialog box. Creating a template library is a two-step process. 1. First, you create a RAD Studio project to use as the basis for the template, and an XML file with a `.bdstemplatelib` extension that describes the project. This project can be any kind of project that RAD Studio supports. 2. Next, you add the project to the list of... more (↗ see page 50)
Overview of Virtual Folders (↗ see page 50)	For C++ only, the **Project Manager** allows any file entry in the project to be arranged and displayed in an arbitrary grouping of your choice called a *virtual folder*. These folders persist in the project file and make no reference to the file's actual location on disk. Virtual folders can only contain file system entries or other virtual folders. Virtual folders can be reordered within the project, be renamed, and be deleted. Deleting a virtual folder does **not** delete the contained files--they simply resume their normal **Project Manager** location prior to their inclusion in the virtual folder. Note that... more (↗ see page 50)
Help on Help (↗ see page 51)	This section includes information about the: • RAD Studio Help • Microsoft .NET Framework SDK Help • CodeGear Developer Support Services and Web Sites • RAD Studio *Quick Start* Guide • Typographic Conventions Used in the Help
Code Completion (↗ see page 53)	Code Completion is a Code Insight feature available in the Code Editor. Code Completion displays a resizable “hint” window that lists valid elements that you can select to add to your code. You can control the sorting of items in the Code Completion hint window by right-clicking the box and choosing **Sort by Name** or **Sort by Scope**. Different items appear in different colors in the list. For example, by default, procedures are teal, functions are dark blue, and abstract methods are shown in red. Automatic code completion is on by default. Options for enabling and disabling Code Completion... more (↗ see page 53)

1.4.1 What is RAD Studio?

RAD Studio is an integrated development environment (IDE) for building Delphi Win32 applications. The RAD Studio IDE provides a comprehensive set of tools that streamline and simplify the development life cycle. The tools available in the IDE depend on the version of RAD Studio you are using. The following sections briefly describe these tools.

Modeling Applications

Modeling can help you can improve the performance, effectiveness, and maintainability of your applications by creating a detailed visual design before you ever write a line of code. RAD Studio provides the **Together** modeling tool, which works with the IDE.

Designing User Interfaces

On the RAD Studio visual designer surface, you can create graphical user interfaces by dragging and dropping components from the **Tool Palette** to a form. Using the designers, you can create Windows Forms applications that use the extensive Visual Component Library (VCL). You can also customize your applications for different versions of Windows, including Windows Vista.

Generating and Editing Code

RAD Studio auto-generates much of your application code as soon as you begin a project. To help you complete the remaining application logic, the text-based **Code Editor** provides features such as refactoring, synchronized editing, code completion, recorded keystroke macros, and custom key mappings. Syntax highlighting and code folding make your code easier to read and navigate.

Compiling, Debugging, and Deploying Applications

Within the IDE, you can set compiler options, compile and run your application, and view compiler messages. RAD Studio integrates MSBuild as a build engine, and both the compile and build commands invoke MSBuild. You can explicitly run MSBuild either by using the command line, or by using the **RAD Studio Command Prompt** on the Start menu. The RAD Studio Command Prompt opens a command console window and automatically sets the path to point to the MSBuild excecutable and sets the environment variable BDS to point to your installation directory.

Compiler options, and several other **Project▶Options**, can be saved as named build configurations, which you can apply to specific projects. By default, the IDE provides a **Debug** and a **Release** build configuration.

The integrated Win32 debugger helps you find and fix runtime and logic errors, control program execution, and step through code to watch variables and modify data values. RAD Studio includes InstallAware for creating Windows Installer setups.

See Also

Designing User Interfaces (see page 15)

Code Editor (see page 42)

Compiling and Building Applications (see page 2)

Debugging Applications (see page 10)

Deploying Applications (see page 19)

1.4.2 What's New in RAD Studio (Delphi for Microsoft .NET)

RAD Studio provides key new features for developing Delphi for Microsoft .NET applications.

Delphi for Microsoft .NET

The following key features are new or significantly changed:

- This release includes a new .NET personality that provides full support for ASP.NET and VCL.NET, as well as inline updates for Delphi 2007 for Win32 and C++Builder 2007.
- Generics (also known as parameterized types) are now supported by the Delphi for .NET compiler. Generics allow you to define a class or record without specifying all the data types to be used within it. You then supply the unspecified types as parameters when you create an instance of the generic type. You can create and consume generics in Delphi for .NET, although the debugger does not evaluate generics. For more information, see Overview of Generics (see page 596).
- **For C#**, functionality has been rolled back to match that for **Visual Basic for .NET**. That is, you can open, edit, compile, and do basic debugging of C# applications. However, the IDE does not offer design-time support for C#.
- Refactorings for Delphi for .NET will not support generics, and Delphi does not support refactoring C# code.

- Template libraries (“starter kits”) enable you to add your own project types to the **New Items** gallery. A template library lets you declare how your project will look. Sample template libraries are included in RAD Studio, including:
- A template library for creating an ASP.NET project that uses ASP.NET 2.0 AJAX Extensions 1.0. This template library is further described under the ASP.NET heading in this topic.
- A template library that supports inserting HTML/CSS layouts such as 3-column, 2-column, header, footer, and so on. For more information about template libraries, see Template Libraries Overview.
- Live Templates support ASP.NET controls. For more information, see Using Live Templates (↗ see page 148).
- Your projects can have a Project Page, which is an HTML file that contains a description of the project, along with various other notes and information that you want to add. The project page is displayed when you open the project in the IDE. You set the Project Page by selecting **Project ▶ Project Page Options**.
- Windows forms are no longer supported in RAD Studio.

ASP.NET

This implementation supports .NET 2.0.

You can create both ASP.NET 2.0 applications, and ASP.NET-based Web Services application. ASP.NET supports the Code-Behind model only.

The Designer provides **MasterPages** and design-time support for all standard ASP.NET 2.0 controls.

You can drag and drop data connections from the Data Explorer to the WebForms Designer.

Site navigation tools will be provided, such as sitemaps, sitemap paths, and sitemap menus.

User controls are supported.

Also supported is login functionality – membership, roles, and associated providers.

ECO support is included for ASP.NET.

An AJAX template library now appears in the **File ▶ New** dialog box. The AJAX library is called **AJAX-Enabled ASP.NET Web Application**. Use this gallery item to create a new ASP.NET project that is set up for use with ASP.NET 2.0 AJAX Extensions 1.0. For more information about Microsoft's ASP.NET AJAX technology, see http://www.asp.net/ajax/.

You can use both the Cassini web server and the IIS webserver while developing your project and you can run your application using F9 when using Cassini.

RAD Studio also provides a DBX4 implementation of the ASP.NET Provider model, which allows you to use a DBX4 database for storing membership, roles, session information, and so forth.

Debugger

The following key features are new or significantly changed:

- You can specify whether Microsoft Managed Debug Assistants (MDAs) will display notifications in the event log. For more information, see Event Log Options (↗ see page 995).
- The debugger can now handle components and classes that contain the new generic types but cannot evaluate generics. For more information about generics, see Overview of Generics (↗ see page 596).
- The **Debug Inspector** has a new context menu command, **Bind to Object**, which binds the Inspector to the specific object. For more information, see Debug Inspector (↗ see page 942).

Together Modeling

Support for the Together modeling tool has the following new features:

- Together modeling features have been updated to sujpport generic types in Delphi for .NET.
- Refactorings that depend on Together will not work for code that contains generic types.

Database

Many changes have been made to improve support for database application development in RAD Studio.

Active Query Builder

The SQL window available from the context menu of the **Data Explorer** is now a separate product, **Active Query Builder**, which provides a full set of visual query-building features. For documentation on the Active Query Builder, see http://www.activequerybuilder.com/hs15.html.

dbExpress

dbExpress has the following new features:

- dbExpress has been entirely rewritten in Delphi.
- A new API is available. This allows you to connect to databases. It also provides a framework for writing dbExpress drivers.
- The dbExpress VCL component's implementation has changed with minimal change to the API. Most applications are not affected by changes to the dbExpress VCL. However, there are some new methods, properties, events, constants, and enums.

For more information about changes in dbExpress, see dbExpress 4 New Feaure Overview.

dbExpress Metadata Improvements

The DBX3 metadata was not rich enough for database tooling and did not support all of the metadata types expected from an ADO.NET 2.0 driver. A new MetaData providers architecture provides much greater capability.

- New metadata providers for 9 different database backends are available.
- Each provider is composed of a separate metadata reader and writer implementation.
- The MetaData providers are detached from the driver so that one metadata provider can be used for multiple driver implementations.
- A new unit DBXMetaDataNames provides classes to describe and provide access to metadata.
- dbExpress exposes a DbxMetaDataProvider class to write metadata.

ADODbx Client

This implements the ADO.NET 2.0 interface. It replaces BDP.NET, based on ADO.NET 1.1, which is being deprecated. AdoDbx Client exposes any existing dbExpress 4 driver as an ADO.NET 2.0 provider.

Instructions on connecting and deploying this provider have been included in the summary documentation at the top of the Borland.Data.AdoDbxClientProvider.pas unit as well as in the product documentation.

DBXClient Driver

DBXClient is a DBX4 driver that remotes the DBX4 framework interface over a pluggable network based transport. In this release, a TCP/IP transport is supported.

In this release, DBXClient can only connect to Blackfish SQL.

Blackfish SQL

Blackfish SQL is the Delphi for .NET version of JBuilder's JDataStore. Blackfish SQL is a high-performance transactional database for the .NET platform, and also supports the .NET Compact Framework. Blackfish SQL will hook to ADO.NET 2.0, and to any DBX4 driver.

Blackfish SQL exposes an API that is documented in a companion help volume. For a high-level summary, see Blackfish SQL Overview.

The implementation is in the unit **Borland.Data.DataStore**.

Blackfish SQL is written entirely in C# and provides the following features and benefits:

- Support for industry standards
- Transaction management
- Support for use of managed code in stored procedures, user-defined functions (UDFs), and triggers
- Simple deployment
- Database portability
- High performance and reliability
- Developer tool integration

ASP.NET Provider

These providers implement the ASP.NET provider model in dbExpress 4 and Blackfish SQL and provide support for management of machine.config and web.config for updating the provider assemblies. They also provide tooling for creating a new instance of the provider database for a website.

DBXProvider and Blackfish SQLProvider

These providers implement the ASP.NET provider model in DBX4 and SQLDataStore and will provide support for management of machine.config and web.config for updating the provider assemblies. They also provide tooling for creating a new instance of the provider database for a website.

VCL.NET

New features:

- Support for .NET 2.0 and 64-bit managed components.
- Support for components, classes, methods, and properties that are compatible with the look and feel of the Microsoft Vista operating system.

New VCL components: The following new classes have been added to the Visual Component Library:

- TDBXConnectionEx
- TDBXCursorValue
- TDBXMemoryConnectionFactory
- TDBXMetaDatabaseColumnNames
- TDBXMetaDataCommandsEx
- TDBXMetaDataCommands
- TDBXMetaDataTableTypesEx
- TDBXPropertyNames
- TDBXPropertyNamesEx
- TDBXStreamReader
- TDBXByteStreamReader
- TDBXLookAheadStreamReader
- TDBXValueTypeFlagsEx
- TDBXWideStringValueEx
- TDBXDatabaseMetaDataEx

ECO

ECO now supports VCL.NET in addition to ASP.NET.

ECO online help is now separate from RAD Studio online help.

1.4.3 What's New in RAD Studio (C++Builder)

RAD Studio provides key new features for developing C++ applications.

C++Builder 2007

The following key features are new or significantly changed:

- **MSBuild is the new build engine:** When you build a C++ project, MSBuild now performs the build process. The structure of the project file has also changed to XML and now contains the options needed for MSBuild. The project file extension has changed to `.cbproj`. You can build projects from the command line using the MSBuild command syntax. For more information, see MSBuild Overview.
- **The Project Options dialog box has been reorganized:** New pages have been added to the **Project▶Options** dialog box, and some existing pages have been renamed in order to better organize the options. New options have also been added, such as `–Vb` options that support C++ constructs that are no longer supported in the standard. The new **Project Properties** page allows you to specify that the C++ compiler is to manage library paths, verify package imports, show header dependencies, or use auto-dependency checking. For more information, see Setting Project Options (↗ see page 162). For information on project options that are no longer available in the IDE, see Unavailable Options (↗ see page 893).
- **You can merge Project Options:** Some project options have a **Merge** check box. When checked, the IDE includes the option values from the current build configuration's immediate ancestor. The options for the current configuration are not actually changed. See Project Options (↗ see page 842) for more information on **Merge**.
- **Build configurations are more extensive:** Build configurations have changed. A build configuration contains the options that you set on many pages of **Project▶Options**. Build configurations store sets of command-line options for build tools such as the compiler, linker, and MSBuild. You can create your own configurations, and there are three default configurations (Base, Debug, and Release). For more information, see Build Configurations Overview (C++) (↗ see page 6).
- **Named option sets are new:** You can create and apply named option sets from the build-related pages of the **Project▶Options** dialog box. Named option sets are saved in files with the extension `.optset`. For more information, see Named Option Sets Overview (↗ see page 7).
- **Build order has changed:** MSBuild builds files according to file type (extension) rather than the user-modifiable order previously used. The new build order is Delphi (`.pas`), C/C++ (`.c`/`.cpp`), assembler (`.asm`), then resource (`.rc`). Within each folder or virtual folder, files are built in order according to their file type. For more information, see Overview of Virtual Folders (↗ see page 50).
- **New Build Configuration Manager activates a build configuration:** Use **Project▶Configuration Manager** to select the build configuration that you want to be active for a selected project or projects. The Configuration Manager replaces the existing way of specifying the active configuration for C++ projects. For more information, see Build Configuration Manager (↗ see page 828).
- **You can specify build events:** You can specify commands to execute at specific points in the build process (pre-link events are new; pre-build and post-build events existed in previous releases). Right-click a buildable file in the **Project Manager** and choose Build Events. For more information, see Build Events Dialog (↗ see page 829).
- **You can create and add .targets files to a project:** A .targets file is an XML file that can contain MSBuild scripts, such as lists of tasks to perform. For more information, see Targets files (↗ see page 8).
- **Location of demo code has changed:** Demo code is now in **MyDocuments\RAD Studio\Demos**. Demos were moved out of the Program Files directory due to Microsoft Vista restrictions.
- **You can compile C++ packages with Delphi:** C++Builder supports compiling design-time packages that contain Delphi source files. However, if any of those Delphi sources make reference to IDE-supplied design-time units such as DesignIntf, DesignEditors, and ToolsAPI that exist in DesignIDE100.bpl, you must take steps to ensure that the references can be

resolved by the C++Builder package. See Compiling C++ Design-Time Packages That Contain Delphi Source (↗ see page 106) to learn how to do this.

Unit testing for C++

Support for Unit Testing is integrated with the DUnit Testing Framework. The DUnit framework is based on the JUnit test framework, and shares much of the same functionality.

You can use the C++Builder Unit Testing wizards to quickly generate skeleton templates for the test project, setup and teardown methods, and basic tests. You can then modify the templates, adding the specific test logic to test your particular methods.

You can run tests using either the Console Test Runner or the DUnit GUI Test Runner. The Console Test Runner directs test output to the Console. The DUnit GUI Test Runner displays test results interactively in a GUI window, with results color-coded to indicate success or failure.

C++Builder Web Services support enhancements

C++Builder Web Services support now includes the following:

- unbounded elements
- optional elements
- nullable elements
- WSDL and schema that import external schemas

These enhancements bring C++Builder Web Services support up-to-date with that of Delphi, enabling your applications to interact with the more robust Web Services like eBay, Amazon, MapPoint, and so forth.

IDE

The following key features are new or significantly changed in the integrated development environment (IDE):

- **Vista and XP Themes:** The IDE now supports Windows Vista and XP themes. Themes are on by default, but you can disable themes for either the IDE or for individual applications. For more information, see IDE on Windows Vista (↗ see page 40).
- **Duplicate file names:** A project is now allowed to contain any number of files with the same name. For example, you can have both the files Common\source1.cpp and Product\source1.cpp in your project. The IDE handles generating the object files so that there is no confusion, and the object from both files is used in building the project.
- **Expanded help about the Memory Manager:** The new memory manager, released with Borland Developer Studio 2006, is documented fully in this release of RAD Studio. Topics include: Configuring the memory manager, Monitoring the memory manager, and Using ShareMem and SimpleShareMem to share the memory manager. The Memory Manager routines and variables are listed under VCL in this topic. For more information, see Memory Management Overview (↗ see page 644).
- **Multi-select in Project Manager:** Hold the CTRL key to multi-select files for the **Open**, **Save**, **Save As**, and **Remove from Project** context-menu commands in the **Project Manager**.
- **New File Browser:** Use **View ▶ File Browser** to invoke the **File Browser** to perform basic file commands or to view a file's SVN status. For more information, see File Browser (↗ see page 1036).
- **New toolbar in the Structure view:** A new toolbar available only for C++ allows you to Sort Alphabetically, Group by Type, Group by Visibility, Show Type, and Show Visibility. For more information, see View>Structure.
- **Virtual folders in the Project Manager:** You can create virtual associations between items in the tree structure. You can use virtual folders to influence the build order. For more information, see Overview of Virtual Folders (↗ see page 50).

Debugger

The following key features are new or significantly changed:

- **Prevent scrolling of the event log:** A new option on the **Tools ▶ Options ▶ Debugger Options ▶ Event Log** page prevents the event log from scrolling new events into view as they occur.
- **CPU windows:** You can now open individual panes of the **CPU** window, such as the **Disassembly**, **CPU Stack**, and **Registers** panes. These single panes of the **CPU** window are dockable; you can undock the panes and dock them elsewhere in the IDE according to your needs. The **CPU** window also now automatically closes when you end the debugging session, and the **Disassembly** pane contains two new options (**Show Opcodes** and **Show Addresses**). For more information, see

CPU Window (see page 1022).

- **Call Stack Window:** You can now set a breakpoint on a particular frame. For more information, see Call Stack Window (see page 1021).
- **Ignore non-user breakpoints:** You can now specify that the debugger is to ignore breakpoints that you did not specifically set using the IDE. For more information, see CodeGear Debuggers (see page 986).
- **Debug Source Path:** The source path for debugging is now a global setting that you create on the **Project**▶**Options**▶**Debugger** page. For more information, see Setting the Search Order for Debug Symbol Tables (see page 129).
- **New toolbar button:** The **Notify on Language Exceptions** command is now an icon on the **View**▶**Toolbars**▶**Customize**▶**Commands** ▶ **Categories** ▶**Run** page. You can click and drag the icon to your toolbar for quick access. For more information, see Language Exceptions (see page 999).
- **Transparent tooltips:** To make a debugger evaluator tooltip transparent, press the `CTRL` key when the tooltip is displayed. Making a tooltip transparent enables you to read the screen behind the tooltip.

Together Modeling

Support for the Together modeling tool is new for C++:

- C++Builder 2007 provides limited modeling support from Together, the fully integrated modeling tool in the IDE. Note that only the code visualization (read-only), documentation generation, and diagram printing features are available in C++ Builder 2007, but the online help describes the full set of Together features.
- **C++ Class Diagrams (Code Visualization):** The C++ class diagram is only available in read-only mode. You can create design diagrams in your C++ projects, but you cannot create classes, interfaces, and so forth in the **Model View**.
- **Design Diagrams:** The complete set of design diagrams are available only in the Enterprise edition of the product. This includes sequence diagrams, collaboration diagrams, state charts, deployment diagrams, use case diagrams, activity diagrams, and component diagrams.
- **Printing diagrams and generating documentation:** Both the Professional and Enterprise editions support the printing of diagrams. The Enterprise edition also supports the generation of documentation.

For more information, see Getting Started With Together (see page 83) or Modeling Applications With Together (see page 81).

Note: Only specific editions of the product contain all the features described in the Together portions of this help system. The current release contains a limited set of features.

Database

Many changes have been made to improve support for database application development in RAD Studio.

dbExpress

Unicode support has been added to the Oracle, Interbase, and MySQL dbExpress drivers.

New driver clients have been added: Interbase 2007 and MySQL 4.1 and 5.

A new dbExpress framework has been created. You can use this framework both to interface with existing drivers and to write new drivers by extending dbExpress framework's abstract classes. You can use the framework directly for both native and managed applications.

A delegate driver is a driver between the application and the actual driver. Delegate drivers allow for pre and post processing of all public methods and properties of the dbExpress 4 framework. Delegate drivers are useful for connection pooling, driver profiling, tracing, and auditing. Sample delegate drivers area provided.

The dbExpress VCL component's API has changed slightly. Most applications are not affected by changes to the dbExpress VCL. However, there are some methods, properties, events, constants, and enums that were removed or replaced with equivalent functionality.

You can also use the dbExpress VCL components that are layered on top of the framework for both native and managed applications. There are some minor API changes in the VCL components to the TSQLConnection class (method changes), TSQLDataSet (new property), and data structures (some are removed or replaced). See dbExpress Framework Compatibility for more information.

The dbExpress driver framework:

- Is written entirely in the Delphi language and allows drivers to be written in Delphi.
- Uses strongly typed data access instead of pointers. For instance, the framework uses String types rather than pointers to strings.
- Is single sourced. This means that a single copy of the source can be compiled with either the native DCC32 or managed DCCIL compilers.
- Has only Abstract base classes that are used for drivers, connections, commands, readers, and so on.
- Uses exception-based error handling rather than returning error codes.

VCL and RTL

New components: The following new components have been added to the Visual Component Library:

AJAX: RAD Studio supports AJAX-based RAD VCL for the Web development.

Microsoft Vista Compatibility: RAD Studio provides components, classes, methods, and properties that are compatible with the look and feel of the Vista operating system.

New VCL components: The following new classes have been added to the Visual Component Library:

- TFileOpenDialog
- TFileSaveDialog
- TTaskDialog
- TCustomFileDialog
- TCustomFileOpenDialog
- TCustomFileSaveDialog
- TCustomTaskDialog
- TFavoriteLinkItem
- TFavoriteLinkItems
- TFavoriteLinkItemsEnumerator
- TFileTypeItem
- TFileTypeItems
- TTaskDialogBaseButtonItem
- TTaskDialogButtonItem
- TTaskDialogButtons
- TTaskDialogButtonsEnumerator
- TTaskDialogProgressBar
- TTaskDialogRadioButtonItem

New Memory Manager routines and variables: The following new System routines and variables have been added to support the Memory Manager:

- AttemptToUseSharedMemoryManager
- GetMemoryManagerState
- GetMemoryMap

- GetMinimumBlockAlignment
- RegisterExpectedMemoryLeak
- SetMinimumBlockAlignment
- ShareMemoryManager
- UnregisterExpectedMemoryLeak
- NeverSleepOnMMThreadContention
- ReportMemoryLeakOnShutdown

See Also

MSBuild Overview (see page 4)

Build Configurations Overview (Delphi) (see page 5)

Build Configurations Overview (C++) (see page 6)

Build Configuration Manager (see page 828)

Project Options (see page 842)

Using the Command Line to Build a Project (see page 108)

Memory Management Overview (see page 644)

File Browser (see page 1036)

Modeling Overview (see page 89)

Targets Files (see page 8)

Overview of Virtual Folders (see page 50)

dbExpress Framework

1.4.4 What's New in RAD Studio (Delphi)

RAD Studio provides key new features for developing Delphi applications for Win32.

IDE

- **MSBuild:** The IDE now supports the MSBuild build engine instead of the previous internal make system. When you open a pre-existing project, the IDE automatically converts the project to the MSBuild format and changes the project extension. You can also use the MSBuild Console (on the Start menu) or MSBuild.exe to build projects from the command line.
- **Build events:** You can specify both DOS commands and macros that are to be performed either before or after compiling your project.
- **Build Configurations:** You can now create named build configurations on the **Project Options** window. To apply a named build configuration to a project or project group, use the new **Build Configuration Manager**, available on the Project menu.
- **Vista and XP Themes:** The IDE now supports Vista and XP themes. Themes are on by default, but you can disable themes for either the IDE or for individual applications.
- **Multi-select in Project Manager:** You can select multiple files for commands such as **Open**, **Save**, **Save As**, and **Remove from Project** in the **Project Manager** context menu.
- **New File Browser:** You can invoke the new **File Browser** to view files on disk and interact with the Windows shell.

Debugger

- **Prevent scrolling of the event log:** A new option on the **Tools**▶**Options**▶**Debugger Options**▶**Event Log** page prevents the event log from scrolling new events into view as they occur.

- **CPU windows:** You can now open individual panes of the **CPU** window, such as the **Disassembly**, **CPU Stack**, and **Registers** panes. These single panes of the **CPU** window are dockable; you can undock the panes and dock them elsewhere in the IDE according to your needs. The **CPU** window also now automatically closes when you end the debugging session, and the **Disassembly** pane contains two new options (**Show Opcodes** and **Show Addresses**).
- **Call Stack Window:** You can now set a breakpoint on a particular frame.
- **Ignore non-user breakpoints:** You can now specify that the debugger is to ignore breakpoints that you did not specifically set using the IDE.
- **Debug Source Path:** The source path for debugging is now a global setting that you create on the **Project▶Options▶Debugger** page.
- **New toolbar button:** The **Notify on Language Exceptions** command is now an icon on the **View▶Toolbars▶Customize▶Categories ▶Run** page. You can click and drag the icon to your toolbar for quick access.
- **Transparent tooltips:** To make a debugger evaluator tooltip transparent, press the `CTRL` key when the tooltip is displayed. Making a tooltip transparent enables you to read the screen behind the tooltip.

Database

Many changes have been made to improve support for database application development in RAD Studio.

dbExpress

Unicode support has been added to the Oracle, Interbase, and MySQL dbExpress drivers.

New driver clients have been added: Interbase 2007 and MySQL 4.1 and 5.

A new dbExpress framework has been created. You can use this framework both to interface with existing drivers and to write new drivers by extending dbExpress framework's abstract classes. You can use the framework directly for both native and managed applications.

A delegate driver is a driver between the application and the actual driver. Delegate drivers allow for pre and post processing of all public methods and properties of the dbExpress 4 framework. Delegate drivers are useful for connection pooling, driver profiling, tracing, and auditing. Sample delegate drivers area provided.

The dbExpress VCL component's API has changed slightly. Most applications are not affected by changes to the dbExpress VCL. However, there are some methods, properties, events, constants, and enums that were removed or replaced with equivalent functionality.

You can also use the dbExpress VCL components that are layered on top of the framework for both native and managed applications. There are some minor API changes in the VCL components to the TSQLConnection class (method changes), TSQLDataSet (new property), and data structures (some are removed or replaced). See dbExpress Framework Compatibility for more information.

The dbExpress driver framework:

- is written entirely in the Delphi language and allows drivers to be written in Delphi.
- uses strongly typed data access instead of pointers. For instance, the framework uses String types rather than pointers to strings.
- is single sourced. This means that a single copy of the source can be compiled with either the native DCC32 or managed DCCIL compilers.
- has only Abstract base classes that are used for drivers, connections, commands, readers, and so on.
- uses exception-based error handling rather than returning error codes.

VCL

AJAX: RAD Studio supports AJAX-based RAD VCL for the Web development.

Microsoft Vista Compatibility: RAD Studio provides components, classes, methods, and properties that are compatible with the look and feel of the Vista operating system.

New components: The following new components have been added to the Visual Component Library:

- TFileOpenDialog
- TFileSaveDialog
- TTaskDialog

New classes: The following new classes have been added:

- TCustomFileDialog
- TCustomFileOpenDialog
- TCustomFileSaveDialog
- TCustomTaskDialog
- TFavoriteLinkItem
- TFavoriteLinkItems
- TFavoriteLinkItemsEnumerator
- TFileTypeItem
- TFileTypeItems
- TTaskDialogBaseButtonItem
- TTaskDialogButtonItem
- TTaskDialogButtons
- TTaskDialogButtonsEnumerator
- TTaskDialogProgressBar
- TTaskDialogRadioButtonItem

See Also

MSBuild Overview (see page 4)

Creating Build Events (see page 107)

Build Configuration Manager (see page 828)

Disabling Themes for the IDE and for the Application (see page 157)

dbExpress Framework Compatibility

File Browser (see page 1036)

1.4.5 Tour of the IDE

When you start RAD Studio, the integrated development environment (IDE) launches and displays several tools and menus. The IDE helps you visually design user interfaces, set object properties, write code, and view and manage your application in various ways.

The default IDE desktop layout includes some of the most commonly used tools. You can use the View menu to display or hide certain tools. You can also customize and save the desktop layouts that work best for you.

The tools available in the IDE depend on the edition of RAD Studio you are using and include the following:

- Welcome Page

- Accessibility Options
- Forms
- Form Designer
- Tool Palette
- Object Inspector
- Object Repository
- Project Manager
- Data Explorer
- Structure View
- History Manager
- Code Editor
- File Browser
- Themes for Windows Vista

The following sections describe each of these tools.

Themes for Windows Vista

The IDE uses Windows Vista or Windows XP themes. If you prefer the classic look in the IDE and in your application, you can turn off theming on the **Project**▶**Options** dialog box.

Welcome Page

When you open RAD Studio, the **Welcome Page** appears with a number of links to developer resources, such as product-related articles, training, and online Help. As you develop projects, you can quickly access them from the list of recent projects at the top of the page. If you close the **Welcome Page**, you can reopen it by choosing **View**▶**Welcome Page**.

Accessibility Options

The IDE's main menu supports MS Active Accessibility (MSAA). This means that you can use the Windows accessibility tools from the Start Menu via **All Programs**▶**Accessories**▶**Accessibility**.

Forms

Typically, a form represents a window or HTML page in a user interface. At design-time, a form is displayed on the Designer surface. You add components from the **Tool Palette** to a form to create your user interface.

RAD Studio provides several types of forms, as described in the following sections. Select the form that best suits your application design, whether it’s a Web application that provides business logic functionality over the Web, or a Windows application that provides processing and high-performance content display. To switch between the Designer and **Code Editor**, click their associated tabs below the IDE.

To access forms, choose **File**▶**New**▶**Other**.

Windows Forms

Use Windows Forms to build native Windows applications that run in a managed environment. You use the .NET classes to build Windows clients, which presents two major advantages—it allows application clients to use features unavailable to browser clients, and it leverages the .NET Framework infrastructure. Windows Forms present a programming model that takes advantage of a unified .NET Framework (for security and dynamic application updates, for instance) and the richness of GUI Windows clients. You use Windows controls, such as buttons, list boxes, and text boxes, to build your Windows applications.

To access a Windows Form, choose **File**▶**New**▶**Other**▶**Delphi for .NET Projects**▶**Windows Forms Application**.

VCL Forms

Use VCL Forms to create native applications using VCL components or to create applications that use VCL.NET components to run in the .NET Framework.

VCL Forms are useful if you want to port an existing Delphi application containing VCL controls to the .NET environment, or if you are already familiar with the VCL and prefer to use it.

You use the classes in the CodeGear Visual Component Library for .NET to create a VCL Forms application.

To access a VCL Forms, choose **File**▶**New**▶**Other**▶**VCL Forms Application**.

ASP.NET Web Forms

Use ASP.NET Web Forms to create applications that can be accessed from any Web browser on any platform. You use the .NET classes to create a ASP.NET Web Forms application. The form consists of the visual representation of the HTML, the actual HTML, and a code-behind file.

To access an ASP.NET Web Form, choose **File**▶**New**▶**Other**▶**Delphi for .NET Projects**▶**Windows Forms Application**.

Form Designer

The **Form Designer**, or Designer, is displayed automatically in the center pane when you are using a form. The appearance and functionality of the Designer depends on the type of form you are using. For example, if you are using an ASP.NET Web Form, the Designer displays an HTML tag editor. To access the Designer, click the **Design** tab at the bottom of the IDE.

Visual Components

Visual components appear on the form at design-time and are visible to the end user at runtime. They include such things as buttons, labels, toolbars, and listboxes.

Form Preview

A preview icon at the bottom right of the Designer (for VCL Forms) shows the positioning of your form as it appears on the screen at runtime. This allows you to position the forms of your application in relation to each other as you design them.

HTML Designer

Use the **HTML Designer** to view and edit ASP.NET Web Forms or HTML pages. This Designer provides a **Tag Editor** for editing HTML tags alongside the visual representation of the form or page. You can also use the **Object Inspector** to edit properties of the visible items on the HTML page and to display the properties of any current HTML tag in the **Tag Editor**. A combo box located above the **Tag Editor** lets you display and edit SCRIPT tags.

To create a new HTML file, choose **File**▶**New**▶**Other**▶**Web Documents**▶**HTML Page**.

Nonvisual Components and the Component Tray

Nonvisual components are attached to the form, but they are only visible at design-time; they are not visible to end users at runtime. You can use nonvisual components as a way to reuse groups of database and system objects or isolate the parts of your application that handle database connectivity and business rules.

When you add a nonvisual component to a form, it is displayed in the component tray at the bottom of the Designer surface. The component tray lets you distinguish between visual and nonvisual components.

Design Guidelines

If you are creating components for a form, you can register an object type and then indicate various points on or near a component's bounds that are "alignment" points. These "alignment" points are vertical or horizontal lines that cut across a visual control's bounds.

When you have the alignment points in place, you can supply UI guideline information so that each component adheres to rules such as distance between controls, shortcuts, focus labels, tab order, maximum number of items (listboxes, menus), etc. In this way, the Form Designer can assist the Code Developer in adhering to established UI guidelines.

If the Snap to Grid option is enabled, and Use Designer Guidelines is also enabled, the designer guidelines take precedence. This means that if a grid point is within the tolerance of the new location and a guideline is also within that distance away, then the control snaps to the guideline instead of the grid position, even if the guideline does not fall on the grid position. The snap tolerance is determined by the grid size. Even if the Snap to Grid and Show Grid options are disabled, the Designer still uses the grid size in determining the tolerance.

This feature is currently only available in VCL and VCL.NET only (This includes C++). See the link at the end of this topic for more information about setting Designer Guidelines.

Tool Palette

The **Tool Palette**, located on the right-hand column, contains items to help you develop your application. The items displayed depend on the current view. For example, if you are viewing a form on the Designer, the **Tool Palette** displays components that are appropriate for that form. You can double-click a control to add it to your form. You can also drag it to a desired position on the form. If you are viewing code in the **Code Editor**, the **Tool Palette** displays code segments that you can add to your application.

Customized Components

In addition to the components that are installed with RAD Studio, you can add customized or third party components to the **Tool Palette** and save them in their own category.

Component Templates

You can create templates that are made up of one or more components. After arranging components on a form, setting their properties, and writing code for them, you can save them as a component template. Later, by selecting the template from the **Tool Palette**, you can place the preconfigured components on a form in a single step; all associated properties and event-handling code are added to your project at the same time. You can reposition the components independently, reset their properties, and create or modify event handlers for them just as if you had placed each component in a separate operation.

Object Inspector

The **Object Inspector**, located on the left, lets you set design time properties and create event handlers for components. This provides the connection between your application's visual appearance and the code that makes the application run. The **Object Inspector** contains two tabs: **Properties** and **Events**.

Use the **Properties** tab to change physical attributes of your components. Depending on your selection, some category options let you enter values in a text box while others require you to select values from a drop-down box. For Boolean operations, you toggle between True or False. After you change your components' physical attributes, you create event handlers that control how the components function.

Use the **Events** tab to specify the events for a specific object you select. If there is an existing event handler, use the drop-down box to select it. By default, some options in the **Object Inspector** are collapsed. To expand the options, click the plus sign (+) next to the category.

Certain nonvisual components, for example, the Borland Data Providers, allow quick access to editors such as the **Connection Editor** and **Command Text Editor**. You can access these editors in the **Designer Verb** area at the bottom of the **Object Inspector**. To open the editors, place your cursor over the name of the editor until your cursor changes into a hand and the editor turns into a link. Alternatively, you can right-click the nonvisual component, scroll down to its associated editor and select it. Note that not all nonvisual components have associated editors. In addition to editors, this area can also display hyperlinks to show custom component editors, launch a web page, and show dialog boxes.

Object Repository

To simplify development, RAD Studio offers pre-designed templates, forms, and other items that you can access and use in your application.

Inside the Object Repository

The **Object Repository** contains items that address the types of applications you can develop. It contains templates, forms, and many other items. You can create projects such as class library, control library, console applications, HTML pages, and many others by accessing the available templates.

To open the **Object Repository** , choose **File**▶**New**▶**Other**. A **New Items** dialog box appears, displaying the contents of the **Object Repository** . You can also edit or remove existing objects from the **Object Repository** by right-clicking the **Object Repository** to view your editing options.

Object Repository Templates

You can add your own objects to the **Object Repository** as templates to reuse or share with other developers. Reusing objects lets you build families of applications with common user interfaces and functionality to reduce development time and improve quality.

You can add a starter project, demo, template, or other useful file to the **Object Repository** and then make it available through the New menu. Choose **Project**▶**Add to Repository**. Select your file. Now when you select the **File**New command, you can choose the file you just added and work with a new copy of it.

RAD Studio allows you to create multiple custom template libraries to use as the basis for creating future projects. Template libraries let you to declare how projects can look, and they enable you to add new types of projects to the **Object Repository**.

Project Manager

A project is made up of several application files. The **Project Manager**, located in the top right-hand column, lets you view and organize your project files such as forms, executables, assemblies, objects, and library files. Within the **Project Manager**, you can add, remove, and rename files. You can also combine related projects to form a project group, which you can compile at the same time.

Add References

You can integrate your legacy COM servers and ActiveX controls into managed applications by adding references to unmanaged DLLs to your project and then browse the types, just as you would with managed assemblies. Choose **Project**▶**Add Reference** to integrate your legacy COM servers or ActiveX controls. Alternatively, right-click the **Reference** folder in the **Project Manager** and click **Add Reference**. You can add other .NET assemblies, COM/ActiveX components, or type libraries using the **Add Reference** feature.

Copy References to a Local Path

During runtime, assemblies must be in the output path of the project or in the Global Assembly Cache (GAC) for deployment. In the **Project Manager**, you can right-click an assembly and use the **Copy Local** setting to copy the reference to the local output path. Follow these guidelines to determine whether a reference must be copied.

- If the reference is to an assembly created in another project, select the **Copy Local** setting.
- If the assembly is in the GAC, do not select the **Copy Local** setting.

Add Web References

You can quickly add a Web Reference to your client application and access the Web Service you want to use. When you add a Web Reference, you are importing a WSDL document into your client application, which describes a particular Web Service.

Once you import the WSDL document, RAD Studio generates all the interfaces and class definitions you need for calling that Web Service. To use the Add Web Reference feature, from your **Project Manager**, right-click the **Web Services** node.

Data Explorer

The **Data Explorer** lets you browse database server-specific schema objects, including tables, fields, stored procedure definitions, triggers, and indexes. Using the context menus, you can create and manage database connections. You can also drag and drop data from a data source to most forms to build your database application quickly.

Structure View

The **Structure View** shows the hierarchy of source code or HTML displayed in the **Code Editor**, or components displayed on the Designer. When displaying the structure of source code or HTML, you can double-click an item to jump to its declaration or location in the **Code Editor**. When displaying components, you can double-click a component to select it on the form.

If your code contains syntax errors, they are displayed in the **Errors** folder in the **Structure View**. You can double-click an error to locate its source in the **Code Editor**.

You can control the content and appearance of the **Structure View** by choosing **Tools**▶**Options**▶**Environment Options**▶**Explorer** and changing the settings.

History Manager

The **History Manager**, located in the center pane, lets you see and compare versions of a file, including multiple backup versions, saved local changes, and the buffer of unsaved changes for the active file. If the current file is under version control, all types of revisions are available in the **History Manager**. The **History Manager** is displayed to the right of the **Code** tab and contains the following tabbed pages:

- The **Contents** page displays current and previous versions of the file.
- The **Diff** page displays differences between selected versions of the file.
- The **Info** page displays all labels and comments for the active file.

You can use the **History Manager** toolbar to refresh revision information, revert a selected version to the most current version, and synchronize scrolling between the source viewers in the **Contents** or **Diff** pages and the **Code Editor** and for window navigation (such as Go to next diff).

Code Editor

The **Code Editor**, located in the center pane, provides a convenient way to view and modify your source code. It is a full-featured, customizable, UTF8 editor that provides refactoring, automatic backups, Code Insight, syntax highlighting, multiple undo capability, context-sensitive Help, Live Templates, Smart Block Completion, Find Class, Find Unit/Import Namespace, and more. Choose the Code Editor link in the section below to view descriptions for each of these Code Editor features.

File Browser

You can perform basic file operations using the **File Browser**, a dockable Windows style browser pane. Open the **File Browser** to search, rename, or perform source control operations on a file.

To perform an operation on a file, choose **View**▶**File Browser**, browse to the file, right-click the file, and select the operation to perform.

See Also

Starting a Project (↗ see page 47)

Form Designer (↗ see page 46)

Adding Components to a Form (↗ see page 152)

Setting Component Properties (↗ see page 161)

Code Editor (↗ see page 42)

Customizing the Code Editor (↗ see page 141)

Building an ASP.NET Application

Disabling Windows Vista or Windows XP Themes (↗ see page 157)

File Browser (↗ see page 1036)

Using the History Manager (↗ see page 149)

1.4.6 IDE on Windows Vista

The IDE includes support for many new Vista user interface features including:

- Vista-style Open File, Save File, and Task dialog boxes.
- Vista theming.
- AERO glass effects in controls.

New Dialogs for Vista

Windows Vista introduces three new types of dialog boxes supported by the IDE. **Task dialogs** are similar to Message dialogs, but have added functionality. **Windows Vista File Dialog and Windows Vista Save Dialog** provide new and changed functionality. RAD Studio provides support for the new Vista dialogs in the **Tool Palette** as well as in the VCL components TTaskDialog, TFileOpenDialog, and TFileSaveDialog.

You can also upgrade existing Message, File, and Save dialogs to the newer Vista dialogs by setting the **UseLatestCommonDialogs** flag. If the flag is set, and you are running Vista, and you have theming enabled, the old-style dialogs will be promoted to Vista-style dialogs. If these conditions are false, Task dialogs are downgraded to Message dialogs. Vista-style File and Save dialogs do not share all of the functionality of the old File and Save dialogs, however, so the IDE provides upgrade capability but no downgrade capability for these controls.

Task dialog boxes include all the functionality of the old Message dialog as well as the following new features and controls:

- A new message title.
- Support for hyperlinks in dialog box.
- User-defined buttons.
- Command link button style.
- Button hints for command link buttons.
- Ordinary progress bars and marquee progress bars.
- Expanded dialog text (**SeeDetails** and **HideDetails**).
- A verification box.
- Controls with elevated security requirements (**ElevationRequired** property).
- Main and footer icons.

Vista-style File and Save dialogs provide similar functionality as the old-style dialogs, but have some substantial changes that can affect existing applications.

The new dialog boxes do not provide user-specified context-sensitive help, but always display Microsoft help. If you need to have application-specific context-sensitive help for your File and Save dialogs, you should continue to use the old-style dialogs.

The Vista File and Save dialogs also have eliminated support for the events **OnShow**, **OnIncludeItem**, and **OnClose**. If your application depends on those events, you should continue to use the old-style dialogs.

Themes in Windows Vista

Windows Vista uses themes in the user interface. For example, TCustomForm is themed for Vista. FileDialog and SaveDialog exist in both classic and themed versions.

Themed versions have some differences from unthemed versions, such as no support for OnShow, IncludeItem, or OnClose events. The new file dialogs do not support custom help but do support Microsoft help.

To upgrade existing projects to use themes, set the Vista flag On. This is a one-way setting; you cannot downgrade to the classic look after upgrading to themes.

By default, themes are On in the IDE and in your application, but you can disable the use of themes.

AERO Glass Effects in Controls

Vista AERO provides an optional glass effect that makes windows and dialogs translucent, so that you can see the graphics below them.

Windows can either have a customizable glass frame or become completely translucent with the **SheetOfGlass** property. Controls in a glassed area can be difficult to see unless you set the **DoubleBuffered** property.

Many controls available in RAD Studio support glass, but some do not.

See Also

MSBuild Overview (see page 4)

Disabling Themes in the IDE and in the Application (see page 157)

1.4.7 Tools Overview

RAD Studio provides several developer tools that are available in the IDE and as standalone executables, as described in the following table.

Tool	Purpose	Executable File Name
Data Explorer	Browse and edit database server-specific schema objects, including tables, fields, stored procedure definitions, triggers, and indexes.	`dbexplor.exe`
XML Mapper	Map nodes in an XML document to fields in a data packet used by a client dataset.	`xmlmapper.exe`
Package Collection Editor	View and edit Delphi packages and other files associated with a package collection (`.dpc` files).	`pce.exe`
Reflection Tool	Inspect types contained within a .NET assembly.	`reflection.exe`

See Also

Data Explorer (see page 1034)

XML Mapper (see page 1011)

Package Collection Editor

1.4.8 Code Editor

The **Code Editor** occupies the center pane of the IDE window. The **Code Editor** is a full-featured, customizable, UTF8 editor that provides syntax highlighting, multiple undo capability, and context-sensitive Help for language elements.

As you design the user interface for your application, RAD Studio generates the underlying code. When you modify object properties, your changes are automatically reflected in the source files.

Because all of your programs share common characteristics, RAD Studio auto-generates code to get you started. You can think of the auto-generated code as an outline that you can examine to create your program.

The **Code Editor** provides the following features to help you write code:

- Change Bars
- Code Insight
- Code Completion
- Code Browsing
- Help Insight
- Live Templates
- Code Folding
- Refactoring
- Sync Edit
- To-Do Lists
- Keystroke Macros
- Bookmarks
- Block comments
- Template Libraries

Change Bars

The left margin of the **Code Editor** displays a green change bar to indicate lines that have not been changed in the current editing session. A yellow change bar indicates that changes have been made since the last File->Save operation.

You can, however, customize the change bars to display in colors other than the default green and yellow.

Code Insight

Code Insight refers to a subset of features embedded in the **Code Editor** (such as Code Parameter Hints, Code Hints, Help Insight, Code Completion, Class Completion, Block Completion, and Code Browsing) that aid in the code writing process. These features help identify common statements you wish to insert into your code, and assist you in the selection of properties and methods. Some of these features are described in more detail in the sub-sections below.

To invoke **Code Insight**, press `CTRL+SPACE` while using the **Code Editor**. A pop-up window displays a list of symbols that are valid at the cursor location.

To enable and configure **Code Insight** features, choose **Tools**▶**Options**▶**Editor Options** and click **Code Insight**.

When you're using the Delphi Language, the pop-up window filters out all interface method declarations that are referred to by property read or write clauses. The window displays only properties and stand-alone methods declared in the interface type. Code insight supports WM_xxx, CM_xxx, and CN_xxx message methods based on like named constants from all units in the uses clause.

Code Parameter Hints

Displays a hint containing argument names and types for method calls. Available between the parenthesis of a call i.e. ShowMessage (|);

You can invoke Code Parameter Hints by pressing `CTRL+SHIFT+SPACE`.

Code Hints

Display a hint containing information about the symbol such as type, file and line # declared at.

You can display Code Hints by hovering the mouse over an identifier in your code, while working in the Code Editor.

Note: Code Hints only work when you have disabled the Help Insight feature.

Help Insight

Help Insight displays a hint containing information about the symbol such as type, file, line # declared at, and any XML documentation associated with the symbol (if available).

Invoke Help Insight by hovering the mouse over an identifier in your code, while working in the Code Editor. You can also invoke Help Insight by pressing `CTRL+SHIFT+H`.

Code Completion

The Code Completion feature displays a drop-down list of available symbols at the current cursor location. You invoke Code Completion for your specific language in the following way:

Delphi — `CTRL + SPACE + .`

C# — `CTRL + SPACE + .`

C++ — `CTRL + SPACE + —>`

Class Completion

Class completion simplifies the process of defining and implementing new classes by generating skeleton code for the class members that you declare. By positioning the cursor within a class declaration in the interface section of a unit and pressing `CTRL+SHIFT+C`, any unfinished property declarations are completed. For any methods that require an implementation, empty methods are added to the implementation section. They are also on the editor context menu.

Block Completion

When you press `ENTER` while working in the Code Editor and there is a block of code that is incorrectly closed, the Code Editor enters the closing block token at the next available empty line after the current cursor position. For instance, if you are using the **Code Editor** with the Delphi language, and you type the token **begin** and then press `ENTER`, the Code Editor automatically completes the statement so that you now have: **begin end;** . This feature also works for the C# and C++ languages.

Code Browsing

While using the **Code Editor** to edit a VCL Form application, you can hold down the `CTRL` key while passing the mouse over the name of any class, variable, property, method, or other identifier. The mouse pointer turns into a hand and the identifier appears highlighted and underlined; click on it, and the **Code Editor** jumps to the declaration of the identifier, opening the source file, if necessary. You can do the same thing by right-clicking on an identifier and choosing Find Declaration.

Code browsing can find and open only units in the project Search path or Source path, or in the product Browsing or Library path. Directories are searched in the following order:

1. The project Search path (**Project ▶ Options ▶ Directories/Conditionals**).

2. The project Source path (the directory in which the project was saved).
3. The global Browsing path (**Tools**▶**Options**▶**Library**).
4. The global Library path (**Tools**▶**Options**▶**Library**▶**Environment Options**▶**Delphi Options** for Delphi).

The Library path is searched only if there is no project open in the IDE.

Code Navigation

The sections below describe features that you can use to navigate through your code while you are using the **Code Editor**.

Method Hopping

You can navigate between methods using a series of editor hotkeys. You can also lock the hopping to occur only within the methods of the current class. For example, if class lock is enabled and you are in a method of TComponent, then hopping is only available within the methods of TComponent.

The keyboard shortcuts for Method Hopping are as follows:

- CTRL+Q^L - toggles class lock
- CTRL+ALT+UP - moves to the top of the current method, or the previous method
- CTRL+ALT+DOWN - moves to the next method
- CTRL+ALT+HOME - first method in source
- CTRL+ALT+END - last method in source
- CTRL+ALT+MOUSE_WHEEL - scrolls through methods

Finding Classes

Allows you to find classes (using C# regular expressions). Use the **Search**▶**Find Class...** command to see a list of available classes that you can select. After you choose one, the IDE navigates to its declaration.

Finding Units

Depending on which language you are programming in, you can use a refactoring feature to locate namespaces or units. If you are using C#, you can use the Use the Import Namespace command to import namespaces into your code. If you are using the Delphi language, you can use the Find Unit... command to locate and add units to your code file. For code that is written using the .NET framework, the Assembly Browser opens if the expression is not found. The Assembly Browser allows you to browse for a type. The **Find Type** window allows regular expressions.

Live Templates

Live Templates allow you to have a dictionary of pre-written code that can be inserted into your programs while you're working with the Code Editor. This reduces the amount of typing that you must do.

Use the links at the end of this topic to learn more about creating and using Live Templates.

Code Folding

Code folding lets you collapse sections of code to create a hierarchical view of your code and to make it easier to read and navigate. The collapsed code is not deleted, but hidden from view. To use code folding, click the plus and minus signs next to the code.

Refactoring

Refactoring is the process of improving your code without changing its external functionality. For example, you can turn a selected code fragment into a method by using the extract method refactoring. The IDE moves the extracted code outside of the current method, determines the needed parameters, generates local variables if necessary, determines the return type, and replaces the code fragment with a call to the new method. Several other refactoring methods, such as renaming a symbol and declaring a variable, are also available.

SyncEdit

The Sync Edit feature lets you simultaneously edit identical identifiers in code. As you change the first identifier, the same change is performed automatically on the other identifiers. You can also set jump points to navigate to specific sections of your code.

To-Do Lists

A **To-Do List** records tasks that need to be completed for a project. After you add a task to the **To-Do List**, you can edit the task, add it to your code as a comment, indicate that it has been completed, and then remove it from the list. You can filter the list to display only those tasks that interest you.

Keystroke Macros

You can record a series of keystrokes as a macro while editing code. After you record a macro, you can play it back to repeat the keystrokes during the current IDE session. Recording a macro replaces the previously recorded macro.

Bookmarks

Bookmarks provide a convenient way to navigate long files. You can mark a location in your code with a bookmark and jump to that location from anywhere in the file. You can use up to ten bookmarks, numbered 0 through 9, within a file. When you set a bookmark, a book icon is displayed in the left gutter of the **Code Editor**.

Block Comments

You can comment-out a section of code by selecting the code in the **Code Editor** and pressing `CTRL+/` (slash). Each line of the selected code is prefixed with `//` and is ignored by the compiler. Pressing `CTRL+/` adds or removes the slashes, based on whether the first line of the code is prefixed with `//`. When using the Visual Studio or Visual Basic key mappings, use `CTRL+K+C` to add and remove comment slashes.

Custom Template Libraries

RAD Studio allows you to create multiple custom template libraries to use as the basis for creating future projects. Template libraries let you to declare how projects can look, and they enable you to add new types of projects to the **Object Repository**.

See Also

Customizing Code Editor (see page 141)

Using Code Insight (see page 146)

Using Class Completion (see page 145)

Using Live Templates (see page 148)

Creating LiveTemplates (see page 138)

Using Sync Edit (see page 150)

Using Code Folding (see page 137)

Using To-Do Lists (see page 166)

Recording a Keystroke Macro (see page 142)

Using Bookmarks (see page 145)

Creating Template Libraries (see page 138)

1.4.9 Form Designer

The **Form Designer** or Designer, is displayed automatically when you are creating or editing a form. To access the Designer, click the **Design** tab at the bottom of the main editing window.

The appearance and functionality of the Designer depends on the type of form you are creating or editing. For example, if you are using an HTML Element, you can display the HTML Tag Editor in the Designer by selecting **View▶Tag Editor**.

Visual Components

You can add Visual components to your form, by dragging them from the **Tool Palette**, in the lower-right section of the IDE, onto the form you are creating. These are the components that will be visible to the end user at runtime. The objects on the **Tool Palette** change dynamically, depending on the type of application or form you are designing.

The tool palette includes controls such as buttons, labels, toolbars, and listboxes for each of the various tool categories; types of applications if you're working at the project level, such as DLL wizards, console or logo applications; and web controls, HTML elements, and data components when you're working on a web application.

Form Preview

A preview icon at the bottom right of the Designer (for VCL Forms) shows the positioning of your form as it will appear on the screen at runtime. This allows you to position the forms of your application in relation to each other as you design them.

Nonvisual Components and the Component Tray

Nonvisual components are attached to the form, but they are only visible at design-time; they are not visible to end users at runtime. You can use nonvisual components as a way to reuse groups of database and system objects or isolate the parts of your application that handle database connectivity and business rules.

When you add an nonvisual component to a form, they are displayed in the component tray at the bottom of the Designer surface. The component tray lets you distinguish between visual and nonvisual components.

HTML Designer

Use the **HTML Designer** to view and edit ASP.NET Web Forms or HTML pages. You can change the default layout in the HTML Designer to be either 'Grid Layout' or 'Flow Layout'. Choose **Tools▶Options** and then select HTML/ASP.NET Options from the tree on the left side. Now you will see the Default Page Layout Options that allows you to select either the Grid Layout or Flow Layout radio button option. This Designer provides a **Tag Editor** for editing HTML tags alongside the visual representation of the form or page. You can also use the **Object Inspector** to edit properties of the visible items on the HTML page and to display the properties of any current HTML tag in the **Tag Editor**. A combo box located above the **Tag Editor** lets you display and edit SCRIPT tags.

To create a new HTML file, choose **File▶New▶Other▶Web Documents▶HTML Page**.

Design Guidelines

If you are creating components for a form, you can register an object type and then indicate various points on or near a component's bounds that are "alignment" points. These "alignment" points are vertical or horizontal lines that cut across the bounds of a visual control.

When you have the alignment points in place, you can supply UI guideline information so that each component will adhere to rules such as distance between controls, shortcuts, focus labels, tab order, maximum number of items (listboxes, menus), etc. In this way, the Form Designer can assist the Code Developer in adhering to established UI guidelines.

If the Snap to Grid option is enabled, and Use Design Guidelines is also enabled, the design guidelines will take precedence. This means that if a grid point is within the tolerance of the new location and a guideline is also within that distance away, then the control will snap to the guideline instead of the grid position, even if the guideline does not fall on the grid position. The snap tolerance is determined by the grid size. Even if the Snap to Grid and Show Grid options are disabled, the Designer will still use the grid size in determining the tolerance.

This feature is currently only available in VCL and VCL.NET only (This includes C++). See the link at the end of this topic for more information about setting up Design Guidelines.

See Also

Starting a Project (see page 47)

Tour of the IDE (see page 34)

Adding Components to a Form (see page 152)

Setting Component Properties (see page 161)

Using Design Guidelines (see page 165)

Building an ASP.NET Application

1.4.10 Starting a Project

A project is a collection of files that is used to create a target application. This collection consists of the files you include and modify directly, such as source code files and resources, and other files that RAD Studio maintains to store project settings, such as the `.dproj` project file. Projects are created at design time, and they produce the project target files (.exe, .dll, .bpl, etc.) when you compile the project.To assist in the development process, the **Object Repository** offers many pre-designed templates, forms, files, and other items that you can use to create applications.

To create a project, click **New** from the **Welcome Page** and select the type of application you want to create, or choose **File▶New▶Other**. To open an existing project, click **Project** from the **Welcome Page** or choose **File▶Open Project**.

This section includes information about

- Types of projects
- Working with unmanaged code

Type of Projects

Depending on the edition of RAD Studio that you are using, you can create traditional Windows applications, ASP.NET Web applications, ADO.NET database applications, Web Services applications, and many others. RAD Studio also supports assemblies, custom components, multi-threading, and COM. For a list of the features and tools included in your edition, refer to the feature matrix on either the CodeGear Delphi web page or the CodeGear C#Builder web page.

Windows Applications

You can create Windows applications using the VCL to provide processing and high-performance content display. In addition to traditional uses for Windows applications, a Windows application can be used with constructs from the .NET framework. For instance, a Windows application can function as a front end to an ADO.NET database.

ASP.NET Web Applications

You can create Web applications using ASP.NET Web Forms to provide Web access to databases and Web Services. Web Forms provide the user interface for Web applications and consist of HTML, server controls, and application logic in code-behind files. RAD Studio lets you drag and drop components and provides in-place HTML editing.

In addition to drag and drop components and visual designers, CodeGear provides an easy way to create application menus and submenus. The .NET Menu Designers MainMenu and ContextMenu are components that work like editors to let you visually design menus and quickly code their functionality.

ASP.NET Web Services Applications

You can create Web Services applications that deliver content, such as HTML pages or XML documents, over the Web. Web Services is an Internet-based integration methodology that allows applications to connect through the Web and exchange information using standard messaging protocols.

RAD Studio simplifies the creation of Web Services by providing methods for creating a SOAP Server application. The `.asmx` and `.dll` files are created automatically, and you can test the Web Service within the IDE, without writing a client application for it.

When writing a client application that uses, or *consumes*, a published Web Service, you can use the UDDI Browser to locate and import WSDL that describes the Web Service into your client application.

VCL.NET Applications

You can use VCL Forms to create a .NET Windows application that uses components from the VCL.NET framework.

RAD Studio simplifies the task of building .NET-enabled applications by supporting VCL components that have been augmented to run on the .NET Framework. This eliminates the need for you to create custom components to provide standard VCL component capabilities. This makes the process of porting Win32 applications to .NET much simpler and more reliable.

Database Applications

Whether your application uses Web Forms or VCL Forms, RAD Studio has several tools that make it easy to connect to a database, browse and edit a database, execute SQL queries, and display live data at design time.

dbExpress allows you to connect to Interbase, Oracle, MS SQL, Informix, DB2, Sybase, and MySQL databases. You can also write database drivers by extending the classes in the dbExpress framework. You can use both native and managed code.

The ADO.NET framework data providers let you access MS SQL, Oracle, and ODBC and OLE DB-accessible databases. The Borland Data Providers (BDP.NET) let you access MS SQL, Oracle, DB2, and InterBase databases. You can connect to any of these data sources, expose their data in datasets, and use SQL commands to manipulate the data. Using BDP.NET provides the following advantages:

- Portable code that's written once and connects to any supported database.
- Open architecture that allows you to provide support for additional database systems.
- Logical data types that map easily to .NET native types.
- Consistent data types that map across databases, where applicable.
- Unlike OLE DB, there is no need for a COM/Interop layer.

When using VCL Forms and the VCL.NET framework components, you can extend database support even further by using the BDE.NET, dbExpress.NET, and Midas Client for .NET connection technologies.

Model-Driven Applications

Modeling is a term used to describe the process of software design. Developing a model of a software system is roughly equivalent to an architect creating a set of blueprints for a large development project. Like a set of blueprints, a model not only depicts the system as a whole, but also allows you to focus in on specifics such as structural and behavioral details. Abstracted away from any particular programming language (and at some levels, even from specific technology), the model allows all participants in the development cycle to communicate in the same language.

CodeGear's Model Driven Architecture (MDA) describes an approach to software engineering where the modeling tools are completely integrated within the development environment itself. The MDA is designed around CodeGear’s Enterprise Core Objects (ECO) framework. The ECO framework is a set of interface, classes, and custom attributes that provide the communication conduit between your application and the modeling-related features of the IDE.

The ECO features include:

- Automatic mapping of the model classes, with their attributes and relationships, to a relational schema.
- Automatic evolution of schema when the model changes.
- Specification of the persistence backend. You can choose to store objects in a relational database or in an XML file.
- Design-time structural validation of the model and its Object Constraint Language (OCL) expressions.
- Runtime validation of the OCL expressions.
- An event mechanism that allows you to receive notifications whenever objects are added, changed, or removed.

RAD Studio IDE leverages the ECO framework to provide an integrated surface on which to develop your application model. The IDE and its modeling surface features include:

- Creating model-driven applications as a new kind of project.
- Creating class diagrams, and manipulating model elements (packages, and classes) directly on the surface.
- Adding, removing, and changing class attributes and methods on the class diagram.
- Two-way updating between source code and the modeling surface. Changes in source code are reflected in the graphical depiction, and vice versa.
- Two-way navigating between model elements and source code. You can navigate from the graphical depiction of a model element directly to its corresponding source code. Similarly, you can navigate from a modeled class in source code directly to its graphical diagram on the modeling surface.
- Exporting and importing models using XMI 1.1.

Note: Not all modeling features are available in all editions of RAD Studio. To determine the modeling features supported in your product edition, refer to the feature matrix on either the CodeGear Delphi web page or the CodeGear C#Builder web page.

Assemblies

An assembly is a logical package, much like a DLL file, that consists of manifests, modules, portable executable (PE) files, and resources (`.html`, `.jpeg`, `.gif`) and is used for deployment and versioning. An application can have one or more assemblies that are referenced by one or more applications, depending on whether the assemblies reside in an application directory or in a global assembly cache (GAC).

Additional Projects

In addition to the project types described above, RAD Studio provides templates to create class libraries, control libraries, console applications, Visual Basic applications, reports, text files, and more. These templates are stored in the **Object Repository** and you can access them by choosing **File ▸ New ▸ Other**.

Unmanaged Code and COM/Interop

Unmanaged code refers to applications that do not target the .NET Framework Common Language Runtime (CLR). COM/Interop is a .NET service that allows seamless interoperation between managed and unmanaged code. The COM/Interop service allows you to leverage existing COM servers and ActiveX controls in your .NET applications, and expose .NET components in legacy unmanaged applications. The RAD Studio IDE includes tools to help you integrate your legacy COM servers and ActiveX controls into managed applications. Additionally, you can add references to unmanaged DLLs to your project, and then browse the types contained, just as you would with managed assemblies.

See Also

Creating a Project (see page 155)

Building an ASP.NET Application

Building an ASP.NET "HelloWorld" Web Services Application

Adding References to a COM Server

1.4.11 Template Libraries

RAD Studio allows you to create multiple custom template libraries to use as the basis for creating future projects. Template libraries let you declare how a project can look, and enable you to add new types of projects to the **New Items** dialog box.

Creating a template library is a two-step process.

1. First, you create a RAD Studio project to use as the basis for the template, and an XML file with a `.bdstemplatelib` extension that describes the project. This project can be any kind of project that RAD Studio supports.
2. Next, you add the project to the list of template libraries in the IDE by pointing to the `.bdstemplatelib` template library file in the **Template Libraries** dialog box, accessed with **Tools**▶**Template Libraries**.

Adding a template library does not create a project. It simply adds metadata information to the development environment that tells the IDE how you create this kind of project, and it adds an icon for it to the specified **New Items** dialog box page. Once the custom template library is in the **New Items** dialog box, you can use it to create a new project.

You can create your own template libraries, and you can use those created by other developers. RAD Studio delivers a default template library which cannot be removed, however you can add and remove custom template libraries.

Note: When creating a project to use with a template library, make sure the project is located in a subdirectory that contains no other projects. All of the files that are in the project should be located within the subdirectory or child subdirectories.

You can either create a single `.bdstemplatelib` template library file for each template library project, or list several related template projects in the same `.bdstemplatelib` template library file by assigning each project a separate unique item number.

See Also

Creating Template Libraries (see page 138)

1.4.12 Overview of Virtual Folders

For C++ only, the **Project Manager** allows any file entry in the project to be arranged and displayed in an arbitrary grouping of your choice called a *virtual folder*. These folders persist in the project file and make no reference to the file's actual location on disk.

Virtual folders can only contain file system entries or other virtual folders. Virtual folders can be reordered within the project, be renamed, and be deleted. Deleting a virtual folder does **not** delete the contained files--they simply resume their normal **Project Manager** location prior to their inclusion in the virtual folder.

Note that changing the order of entries in a virtual folder changes the build order of the contained buildable entries. In general, the order files appear in the project manager specifies the order in which they are built. All files that are processed by a tool are built in batches — each type is done separately. The file type processing order is:

- Delphi (.pas)
- C/C++ (.c/.cpp)
- Assembler (.asm)
- Resource (.rc)

In other words, all Delphi files are compiled first, then C/C++ files in the order they appear in the project manager, and so on.

You can drag any file entry in the **Project Manager** into and out of any virtual folder in the project. You can also right-click a virtual folder and use the context menu commands to add items to the virtual folder.

See Also

Using Virtual Folders (see page 167)

1.4.13 Help on Help

This section includes information about the:

- RAD Studio Help
- Microsoft .NET Framework SDK Help
- CodeGear Developer Support Services and Web Sites
- RAD Studio *Quick Start* Guide
- Typographic Conventions Used in the Help

RAD Studio Help

The RAD Studio Help includes conceptual overviews, procedural how-to's, and reference information, allowing you to navigate from general to more specific information as needed.

Additionally, the persistent navigation panes in the Help window make it easier to locate and filter information. By default, no filter is set, allowing you to view all of the installed Help. However, to narrow the focus when searching the Help or using the index, use the **Filter by:** drop-down list on the **Content**, **Search**, and **Index** panes. To display the navigation panes, use the **View▶Navigation** menu command.

Tip: When navigating to a topic by using a link from another topic, the context of the topic you are viewing might not be obvious. To find the context of that topic within the Content

pane, click the **Sync Contents** ⇔ button on the toolbar of the **CodeGear Help** viewer.

Conceptual Overviews

The conceptual overviews provide information about product architecture, components, and tools that simplify development. If you are new to a particular area of development, such as modeling or ADO.NET, see the overview topic at the beginning of each section in the online Help.

At the end of most of the overviews, you will find links to related, more detailed information. Icons are used to indicate that a link leads to the .NET SDK, partner Help, or to a web site. The icons are explained later in this topic.

Procedures (How-To)

The how-to procedures provide step-by-step instructions. For development tasks that include several subtasks, there are *core procedures*, which include the subtasks required to accomplish a larger task. If you are beginning a development project and want to know what steps are involved, see the core procedure for the area you are working on. In addition to the core procedures, there are several single-task procedures.

All the procedures are listed under **Procedures** in the **Content** pane of the Help window. Additionally, most of the conceptual overviews provide links to the pertinent procedures.

Reference Topics

The reference topics provides detailed information on subjects such as API elements, the Delphi language, and compiler directives.

All of the reference topics are located under **Reference** in the **Content** pane of the Help window. Additionally, most API references are underlined and link directly to the appropriate reference topic.

Context Sensitive F1 Help

Context sensitive Help is available throughout the IDE by selecting an item and pressing `F1`:

- In the **Code Editor**, select and highlight the entire element, such as a namespace, keyword, or method
- On a form **Design** tab, select the component
- In the **Messages** window, select a message
- Within IDE windows, such as the **Project Manager** or **Model View**, click within the window

  Note: Pressing F1

  on an element that is part of the VCL.NET framework displays the RAD Studio Help. Pressing `F1` on an element that is part of the .NET framework displays the Microsoft .NET Help.

Microsoft SDK Help

RAD Studio is distributed with the both the Microsoft .NET Framework SDK and the Microsoft Platform SDK, which include extensive online Help. Where appropriate, the RAD Studio Help provides links to the SDK online Help. Alternatively, you can access the SDK Help directly from the **Content** pane of this Help system.

CodeGear Developer Support Services and Web Site

CodeGear offers a variety of support options to meet the needs of its diverse developer community. To find out about support, refer to www.borland.com/devsupport. From the web site, you can access many newsgroups where developers exchange information, tips, and techniques. The site also includes a list of books, technical documents, and Frequently Asked Questions (FAQ). Additionally, you can access the CodeGear Developer Network.

RAD Studio Quick Start Guide

The RAD Studio*Quick Start* guide provides an overview of the RAD Studio development environment to help you install and start using the product right away. The *Quick Start* guide is shipped along with your product.

Typographic Conventions Used in the Help

The following typographic conventions are used throughout the RAD Studio online Help.

Typographic conventions

Convention	Used to indicate
`Monospace type`	Source code and text that you must type.
Boldface	Reserved language keywords or compiler options, references to dialog boxes and tools.
Italics	RAD Studio identifiers, such as variables or type names. Italicized text is also used for book titles and to emphasize new terms.
KEYCAPS	Keyboard keys, for example, the `CTRL` or `ENTER` key.
WEB →	A link to Web resources.
SDK →	An external link to Microsoft SDK documentation.
🤝 →	An external link to documentation provided by CodeGear partners.

See Also

Microsoft Help on Help

Finding Information with the Index

1.4.14 Code Completion

Code Completion is a Code Insight feature available in the Code Editor. Code Completion displays a resizable “hint” window that lists valid elements that you can select to add to your code. You can control the sorting of items in the Code Completion hint window by right-clicking the box and choosing **Sort by Name** or **Sort by Scope**.

Different items appear in different colors in the list. For example, by default, procedures are teal, functions are dark blue, and abstract methods are shown in red.

Automatic code completion is on by default. Options for enabling and disabling Code Completion are located on the **Code Insight** page of the Tools->Options->Editor Options dialog box.

Using Code Completion

Following are ways to use Code Completion in the IDE:

- To display the properties, methods, and events available in a class, press `Ctrl+Space` after the name of a variable that represents either a class instance or a pointer to a class instance.
- To invoke code completion for a pointer type, the pointer must first be dereferenced. For example, type:`self.`in Delphi, or `this->` in C++.
- Type an arrow for a pointer to an object.
- You can also type the name of non-pointer types followed by a period (.) to see the list of inherited and virtual properties, methods, and events.
- For example, in Delphi type: `TRect test; test.` In C++, type: `var test: TRect; <br>: <br>: <br>begin test.`.
- Type an assignment operator or the beginning of an assignment statement, and press `Ctrl+Space` to display a list of possible values for the variable.
- Type a procedure, function, or method call and press Ctrl+Space to display a list of arguments that are valid for assignment to the variable entered. Select a list item followed by an ellipsis (…) to open a second list of related arguments compatible with the variable entered in the assignment statement.
- Type a record (in Delphi) or a structure (in C++) to display a list of fields.
- Type an array property (not a genuine array), and press `Ctrl+Space` to display an index expression.
- In C++, you can also press `Ctrl+Space` on a blank statement line to display symbols from additional RTL units even if they are not used by the current unit.

Browsing to a Declaration

When the Code Completion list is displayed, you can hold down `Ctrl` and click any identifier in the list to browse to its declaration.

Also, if you hover the mouse pointer over the identifier in the Code Editor, a hint window tells where the identifier is declared. You can press `Ctrl`, point to the identifier in the code (it changes to blue underline, by default, and the insertion point changes to a hand pointing), and then click to move to its declaration.

Note: Code Insight works only in the compilation unit. Code Completion supports WM_xxx, CM_xxx, and CN_xxx message methods based on like named constants from all units in the `uses` clause.

Note: For C++, Code Completion features work best when you have already built your application and have created a precompiled header. Otherwise, you need to wait for the compiler to generate the required information. It is recommended that you check the Use pre-compiled headers

option on the Project->Options->Compiler dialog box.

See Also

Code Editor (↗ see page 42)

1.5 Refactoring Applications

Refactoring is a technique you can use to restructure and modify your code in such a way that the intended behavior of your code stays the same. RAD Studio provides a number of refactoring features that allow you to streamline, simplify, and improve both performance and readability of your application code.

Topics

Name	Description
Add Namespace (↗ see page 57)	**Refactor ▶ Import Namespace** Use this dialog box to import namespaces into the using clause of your code file based on objects in code that are contained in that namespace, but for which no namespace has yet been declared.
Refactoring Overview (↗ see page 57)	Refactoring is a technique you can use to restructure and modify your existing code in such a way that the intended behavior of your code stays the same. Refactoring allows you to streamline, simplify, and improve both performance and readability of your application code. Each refactoring operation acts upon one specific type of identifier. By performing a number of successive refactorings, you build up a large transformation of the code structure, and yet, because each refactoring is limited to a single type of object or operation, the margin of error is small. You can always back out of a particular... more (↗ see page 57)
Change Parameters (↗ see page 58)	**Refactor ▶ Change Params** Adds, edits, removes, and rearranges the parameters of a method.
Symbol Rename Overview (Delphi, C#, C++) (↗ see page 59)	Renames identifiers and all references to the target identifier. You can rename an identifier if the original declaration identifier is in your project or in a project your project depends on, in the Project Group. You can also rename an identifier if it is an error identifier, for instance, an undeclared identifier or type. The refactoring engine enforces a few renaming rules: • You cannot rename an identifier to a keyword. • You cannot rename an identifier to the same identifier name unless its case differs. • You cannot rename an identifier from within a dependent project when the project where the original... more (↗ see page 59)
Add or Edit Parameter (↗ see page 60)	Use the **Add Parameter** or **Edit Parameter** dialog box to add a parameter to a method signature or to edit a parameter that you have already added.
Extract Method Overview (Delphi) (↗ see page 60)	Use the Extract Method refactoring operation to change a code fragment into a method whose name describes the purpose of the method. The Extract Method feature analyzes any highlighted code. If that code is not extractable to a method, the refactoring engine warns you. If the method can be refactored, the refactoring engine creates a new method outside of the current method. The refactoring engine then determines any parameters, generates local variables, determines the return type, and prompts the user for a new name. The refactoring engine inserts a method call to the new method in the location of the... more (↗ see page 60)
Declare Field (↗ see page 61)	**Refactor ▶ Declare Field** Use this dialog box to declare a field in your code.

Extract Resource String (Delphi) (see page 62)	Extracting resource strings helps centralize string definitions which can then be more easily translated, if necessary. You can extract string values to resource strings that are defined in the **resourcestring** section of your code file. If there is no **resourcestring** section in your code, the refactoring engine creates one following either the **implementation** keyword or the **uses** list. You cannot create a resource string from the following elements: • **Constants.** For example, `const A = 'abcdefg';` cannot be extracted to a resource string. • **Constants in Parameters.** For example, in `MyProc(A, B:Integer; C: string='test');` the string cannot be extracted to a resource... more (see page 62)
Declare Variable and Declare Field Overview (Delphi) (see page 62)	You can use the Refactoring feature to create variables and fields. This feature allows you to create and declare variables and fields while coding without planning ahead. This topic includes information about: • Declare Variable • Initial Type Suggestion • Declare Field
Declare Variable (see page 64)	**Refactor ▶ Declare Variable** Use this dialog box to declare a local variable in a procedure. If the cursor in the **Code Editor** is not positioned on an undeclared variable, the **Refactor ▶ Declare Variable** command in unavailable.
Extract Method (see page 64)	**Refactor ▶ Extract Method** Turns a selected code fragment into a method. RAD Studio moves the extracted code outside of the current method, determines the needed parameters, generates local variables if necessary, determines the return type, and replaces the code fragment with a call to the new method.
Find References Overview (Delphi, C#, C++) (see page 65)	Sometimes, you may not want to change code, but want to find references to a particular identifier. The refactoring engine provides **Find References**, **Find Local References**, and **Find Declaration Symbol** commands. Both **Find References** and **Find Local References** commands provide you with a hierarchical list in a separate **Find References** window, showing you all occurrences of a selected reference. If you choose the **Find References** command, you are presented with a treeview of all references to your selection in the entire project. If you want to see local references only, meaning those in the active code file, you... more (see page 65)
Change Parameters Overview (Delphi) (see page 66)	Adding or removing a parameter from a function is a commonly performed and tedious programming task. RAD Studio provides the **Change Parameters** refactoring to automate this task. You can use **Change Parameters** to add, remove, and rearrange function parameters. To use this refactoring, select a function name in the **Code Editor** and choose **Refactor ▶ Change Params**. When you use the **Change Parameters** refactoring, the following function signature conflicts can occur: • A descendant class contains an **override** for the function you are refactoring. When you refactor the function, any functions that **override** the refactored function will also be refactored. • A... more (see page 66)
Extract Resource String (see page 66)	**Refactor ▶ Extract Resource String** Use this dialog box to convert the string currently selected in the **Code Editor** to a resource string. The `resourcestring` keyword and the resource string will be added to the implementation section of your code, and the original string will be replaced with the new resource string name.
Find Unit (see page 66)	**Refactor ▶ Find Unit** Use this dialog to locate units and add them to the uses clause of your Delphi code file.
Sync Edit Mode (Delphi, C#, C++) (see page 67)	Sync Edit mode allows you to change all occurrences of an identifier when you change one instance of that identifier. When you enter Sync Edit mode, you can tab to each highlighted identifier in your current **Code Editor** window. If you change an identifier that appears elsewhere in the file, all occurrences transform to whatever you type, character by character.

Refactorings (↗ see page 67)	**View ▶ Refactorings** Performs the listed refactorings.
Undoing a Refactoring (Delphi, C#) (↗ see page 67)	The refactoring engine takes advantage of a versioning mechanism, known as *local striping*, to allow you to undo renames in source code files. The IDE records the current timestamp of each file included in the current refactoring changeset. The timestamp corresponds to a specific local revision of the file. When you select the undo command, the IDE copies the local backup file that matches that timestamp back over the refactored file. The important point to understand is that any changes that you make to the files after the refactoring will also be rolled back when you perform an Undo.... more (↗ see page 67)
Rename Symbol (C++) (↗ see page 68)	**Refactor ▶ Rename** Use this dialog box to specify a new name for the selected symbol before refactoring your code.
Rename <symbol name> (C#) (↗ see page 68)	**Refactor ▶ Rename <symbol name>** Use this dialog box to perform rename refactoring on a symbol, such as a variable, type, field, method, or parameter, currently selected in the **Code Editor**. The first field in the dialog varies based on the type of symbol you are renaming.
Rename <symbol name> (Delphi) (↗ see page 69)	**Refactor ▶ Rename <symbol name>** Use this dialog box to perform rename refactoring on a symbol, such as a variable, type, field, method, or parameter, currently selected in the **Code Editor**. The first field in the dialog varies based on the type of symbol you are renaming.

1.5.1 Add Namespace

Refactor ▶ Import Namespace

Use this dialog box to import namespaces into the using clause of your code file based on objects in code that are contained in that namespace, but for which no namespace has yet been declared.

Item	Description
Search	Prefilled with the selected object. You can enter another object name to display a completely new list of namespaces. If you clear the textbox, the **Matching Results** field displays all available namespaces.
Matching Results	Displays a list of appropriate namespaces for the specified object. One namespace is highlighted as the most likely namespace containing the object. You can, however, select other namespaces, or multi-select namespaces in the list.

1.5.2 Refactoring Overview

Refactoring is a technique you can use to restructure and modify your existing code in such a way that the intended behavior of your code stays the same. Refactoring allows you to streamline, simplify, and improve both performance and readability of your application code.

Each refactoring operation acts upon one specific type of identifier. By performing a number of successive refactorings, you build up a large transformation of the code structure, and yet, because each refactoring is limited to a single type of object or operation, the margin of error is small. You can always back out of a particular refactoring, if you find that it gives you an unexpected result. Each refactoring operation has its own set of constraints. For example, you cannot rename symbols that are imported by the compiler. These are described in each of the specific refactoring topics.

RAD Studio includes a refactoring engine that evaluates and executes the refactoring operation. The engine also displays a preview of what changes will occur in a refactoring pane that appears at the bottom of the **Code Editor**. The potential refactoring operations are displayed as tree nodes, which can be expanded to show additional items that might be affected by the refactoring, if they exist. Warnings and errors also appear in this pane. You can access the refactoring tools from the **Main** menu and from context-sensitive drop down menus.

RAD Studio provides the following refactoring operations:

- Symbol Rename (Delphi, C#, C++)
- Extract Method (Delphi)
- Declare Variable and Field (Delphi)
- Sync Edit Mode (Delphi, C#)
- Find References (Delphi, C#, C++)
- Extract Resourcestring (Delphi)
- Find Unit (Delphi)
- Use Namespace (C#)
- Undo (Delphi, C#)
- Change Parameters (Delphi)

See Also

Symbol Rename Overview (see page 59)

Refactoring Code (see page 143)

Previewing and Applying Refactoring Operations (see page 111)

Sync Edit Mode (see page 67)

Extract Method Overview (see page 60)

Find References Overview (see page 65)

Declare Variable and Declare Field Overview (see page 62)

Extract Resource String Overview (see page 62)

Finding References (see page 141)

Undo Rename (see page 67)

Finding Namespaces and Finding Units (see page 142)

1.5.3 Change Parameters

Refactor▶Change Params

Adds, edits, removes, and rearranges the parameters of a method.

Item	Description
Class	Displays the class in which the selected method is defined.
Method	Displays the method that you are refactoring.
Parameters	Lists information about the parameters declared in the method.
Add	Displays the **Add Parameter** dialog box, which you use to add a parameter to the method signature.
Edit	Displays the **Edit Parameter** dialog box, which you use to edit a parameter that you have created.

Remove	Removes the selected parameter.
Move Up	Moves the selected parameter up in the parameter declaration list.
Move Down	Moves the selected parameter down in the parameter declaration list.

Note: If you remove a parameter, you need to manually remove any method code that uses the removed parameter.

See Also

Change Parameters Overview (↗ see page 66)

Parameter Types (↗ see page 672)

Fundamental Syntactic Elements (↗ see page 701)

1.5.4 Symbol Rename Overview (Delphi, C#, C++)

Renames identifiers and all references to the target identifier. You can rename an identifier if the original declaration identifier is in your project or in a project your project depends on, in the Project Group. You can also rename an identifier if it is an error identifier, for instance, an undeclared identifier or type.

The refactoring engine enforces a few renaming rules:

- You cannot rename an identifier to a keyword.
- You cannot rename an identifier to the same identifier name unless its case differs.
- You cannot rename an identifier from within a dependent project when the project where the original declaration identifier resides is not open.
- You cannot rename symbols imported by the compiler.
- You cannot rename an overridden method when the base method is declared in a class that is not in your project.
- If an error results from a refactoring, the engine cannot apply the change. For example, you cannot rename an identifier to a name that already exists in the same declaration scope. If you still want to rename your identifier, you need to rename the identifier that already has the target name first, then refresh the refactoring. You can also redo the refactoring and select a new name. The refactoring engine traverses parent scopes, searching for an identifier with the same name. If the engine finds an identifier with the same name, it issues a warning.

Rename Method

Renaming a method, type, and other objects is functionally the same as renaming an identifier. If you select a procedure name in the **Code Editor**, you can rename it. If the procedure is overloaded, the refactoring engine renames only the overloaded procedure and only calls to the overloaded procedure. An example of this rule follows:

```
procedure Foo; overload;
procedure Foo(A:Integer); overload;
Foo();
Foo;
Foo(5);
```

If you rename the first procedure *Foo* in the preceding code block, the engine renames the first, third, and fourth items.

If you rename an overridden identifier, the engine renames all of the base declarations and descendent declarations, which means the original virtual identifier and all overridden symbols that exist. An example of this rule follows:

```
TFoo = class
    procedure Foo; virtual;
end;
```

```
TFoo2 = class(TFoo)
    procedure Foo; override;
end;

TFoo3 = class(TFoo)
    procedure Foo; override;
end;

TFoo4 = class(TFoo3)
    procedure Foo; override;
end;
```

Performing a rename operation on *Foo* renames all instances of *Foo* shown in the preceding code sample.

See Also

Refactoring Overview (see page 57)

Extract Method Overview (see page 60)

Renaming a Symbol (see page 112)

1.5.5 Add or Edit Parameter

Use the **Add Parameter** or **Edit Parameter** dialog box to add a parameter to a method signature or to edit a parameter that you have already added.

Item	Description
Parameter Name	Specifies the name of the new parameter. You must enter a legal Delphi identifier.
Data type	Specifies the type of the new parameter.
Matching Results	Lists data types that match the text you enter in the **Data type** field.
Literal Value	Specifies the value for the new parameter. Refactoring uses this literal as the value of the new parameter in existing calls to this procedure.
Parameter Type	Specifies the parameter type.
Value	Passes the parameter by value.
Var	Passes the parameter by reference.
Out	Passes the parameter by reference and discards the initial value.
Const	Passes the parameter by value and disallows assignment to the parameter.

See Also

Change Parameters Overview (see page 66)

Parameter Types (see page 672)

Fundamental Syntactic Elements (see page 701)

1.5.6 Extract Method Overview (Delphi)

Use the Extract Method refactoring operation to change a code fragment into a method whose name describes the purpose of the method. The Extract Method feature analyzes any highlighted code. If that code is not extractable to a method, the refactoring engine warns you. If the method can be refactored, the refactoring engine creates a new method outside of the current method. The refactoring engine then determines any parameters, generates local variables, determines the return type, and prompts the user for a new name. The refactoring engine inserts a method call to the new method in the location of the old method.

There are certain limitations to the extract method refactoring. They include:

- Cannot extract expressions, only statements.
- Cannot extract statements that include a call to **inherited** in Delphi.
- Cannot extract statements that are contained within a **with** statement.
- Cannot extract statements that call a local procedure or function.

If you select an expression and choose the Extract Method command, your selection will be expanded to include the entire statement. If the expression in your statement is used as a result, the extracted code returns a function result in place of the expression.

See Also

Refactoring Overview (↗ see page 57)

Refactoring Code (↗ see page 143)

Renaming a Symbol (↗ see page 112)

1.5.7 Declare Field

Refactor ▶Declare Field

Use this dialog box to declare a field in your code.

Item	Description
Current Class	Displays the class from which the field will be derived.
Field Name	Displays the name of the field to be declared. By default the name you typed in the **Code Editor** is displayed. If you enter an invalid name in this field, such a reserved word, an exclamation icon ❗ is displayed in the dialog box, prompting you to correct it.
Type	Specifies the data type of the field.
Array	Specifies that the field should be declared a an array.
Dimensions	Specifies the dimensions for the array.
Visibility	Specifies the accessibility of the field. If the value in **Field Name** conflicts with an existing field name in the same scope, the **Refactorings** dialog box is displayed, indicating the conflict.

See Also

Refactoring Overview (↗ see page 57)

1.5.8 Extract Resource String (Delphi)

Extracting resource strings helps centralize string definitions which can then be more easily translated, if necessary. You can extract string values to resource strings that are defined in the **resourcestring** section of your code file. If there is no **resourcestring** section in your code, the refactoring engine creates one following either the **implementation** keyword or the **uses** list.

You cannot create a resource string from the following elements:

- **Constants.** For example, `const A = 'abcdefg';` cannot be extracted to a resource string.
- **Constants in Parameters.** For example, in `MyProc(A, B:Integer; C: string='test');` the string cannot be extracted to a resource string.
- **Resource Strings.** For example, `resourcestring A = 'test';` is already a resource string.

See Also

Refactoring Overview (see page 57)

Refactoring Code (see page 143)

1.5.9 Declare Variable and Declare Field Overview (Delphi)

You can use the Refactoring feature to create variables and fields. This feature allows you to create and declare variables and fields while coding without planning ahead. This topic includes information about:

- Declare Variable
- Initial Type Suggestion
- Declare Field

Declare Variable

You can create a variable when you have an undeclared identifier that exists within a procedure block scope. This feature gives you the capability to select an undeclared identifier and create a new variable declaration with a simple menu selection or keyboard shortcut. When you invoke the **Declare Variable** dialog, the dialog contains a suggested name for the variable, based on the selection itself. If you choose to name the variable something else, the operation succeeds in creating the variable, however, the undeclared identifier symbol (Error Insight underlining) remains.

Variable names must conform to the language rules for an identifier. In Delphi, the variable name:

- Cannot be a keyword.
- Cannot contain a space.
- Cannot be the same as a reserved word, such as **if** or **begin**.
- Must begin with a Unicode alphabetic character or an underscore, but can contain Unicode alphanumeric characters or underscores in the body of the variable name.
- In the Delphi language, the type name can also be the keyword **string**.

  Note: The .NET SDK recommends against using leading underscores in identifiers, as this pattern is reserved for system use.

Note: On the dialog that appears when you choose to declare a variable, you can set or decline to set an initial value for the variable.

Initial Type Suggestion

The refactoring engine attempts to suggest a type for the variable that it is to create. The engine evaluates binary operations of the selected statement and uses the type of the sum of the child operands as the type for the new variable. For example, consider the following statement:

```
myVar := x + 1;
```

The refactoring engine automatically assumes the new variable *myVar* should be set to type Integer, provided *x* is an Integer.

Often, the refactoring engine can infer the type by evaluating a statement. For instance, the statement `If foo Then...` implies that *foo* is a Boolean. In the example `If (foo = 5) Then...` the expression result is a Boolean. Nonetheless, the expression is a comparison of an ordinal (*5*) and an unknown type (*foo*). The binary operation indicates that *foo* must be an ordinal.

Declare Field

You can declare a field when you have an undeclared identifier that exists within a class scope. Like the Declare Variable feature, you can refactor a field you create in code and the refactoring engine will create the field declaration for you in the correct location. To perform this operation successfully, the field must exist within the scope of its parent class. This can be accomplished either by coding the field within the class itself, or by prefixing the field name with the object name, which provides the context for the field.

The rules for declaring a field are the same as those for declaring a variable:

- Cannot be a keyword.
- Cannot contain a space.
- Cannot be the same as a reserved word, such as **if** or **begin**.
- Must begin with a Unicode alphabetic character or an underscore, but can contain Unicode alphanumeric characters or underscores in the body of the field name.
- In the Delphi language, the type name can also be the keyword **string**.

  Note: Leading underscores on identifiers are reserved in .NET for system use.

  You can select a visibility for the field. When you select a visibility that is not **private** or **strict private**, the refactoring engine performs the following operations:

- Searches to find all child classes.
- Searches each child class to find the field name.
- Displays a red error item if the field name conflicts with a field in a descendant class.
- You cannot apply the refactoring if it conflicts with an existing item name.

Sample Refactorings

The following examples show what will happen when declaring variables and fields using the refactoring feature.

Consider the following code:

```
TFoo = class
private
  procedure Foo1;
end;
...

implementation

procedure TFoo.Foo1;
begin
  FTestString := 'test';    // refactor TestString, assign field
```

```
end;
```

Assume you apply a Declare Field refactoring. This would be the result:

```
TFoo = class
private
  FTestString: string;
  procedure Foo1;
end;
```

If you apply a Declare Variable refactoring instead, the result is:

```
procedure TFoo.Foo1;
var                            // added by refactor
  TestString: string;    // added by refactor
begin
  TestString := 'test';     // added by refactor
  TestString := 'whatever';
end;
```

See Also

Refactoring Overview (see page 57)

Symbol Rename Overview (see page 59)

Refactoring Code (see page 143)

1.5.10 Declare Variable

Refactor ▶ Declare Variable

Use this dialog box to declare a local variable in a procedure. If the cursor in the **Code Editor** is not positioned on an undeclared variable, the **Refactor ▶ Declare Variable** command in unavailable.

Item	Description
Name	Displays the name of the currently selected variable.
Type	Specifies the type of variable. By default the name you typed in the **Code Editor** is displayed. If you enter an invalid name in this field, such a reserved word, an exclamation icon ❗ is displayed in the dialog box, prompting you to correct it.
Array	Specifies that the variable should be declared as an array.
Dimensions	Specifies the dimensions of the array.
Set Value	Initializes the variable with specified value.

See Also

Refactoring Overview (see page 57)

1.5.11 Extract Method

Refactor ▶ Extract Method

Turns a selected code fragment into a method. RAD Studio moves the extracted code outside of the current method, determines the needed parameters, generates local variables if necessary, determines the return type, and replaces the code fragment with a call to the new method.

Item	Description
Current Method	Displays the name of the method in which the code fragment currently resides.
New method name	Specifies the name to be used for the newly extracted method. **ExtractedMethod** is displayed by default, but you can overtype it.
Sample extracted code	Displays the code that will be generated for the newly extracted method.

See Also

Refactoring Overview (see page 57)

1.5.12 Find References Overview (Delphi, C#, C++)

Sometimes, you may not want to change code, but want to find references to a particular identifier. The refactoring engine provides **Find References**, **Find Local References**, and **Find Declaration Symbol** commands.

Both **Find References** and **Find Local References** commands provide you with a hierarchical list in a separate **Find References** window, showing you all occurrences of a selected reference. If you choose the **Find References** command, you are presented with a treeview of all references to your selection in the entire project. If you want to see local references only, meaning those in the active code file, you can select the **Find Local References** command from the **Search** menu. If you want to find the original declaration within the active Delphi code file, you can use the **Find Declaration Symbol** command. The **Find Declaration Symbol** command is only valid in Delphi and does not apply to C#.

Sample Refactoring

The following sample illustrates how the Find References refactoring will proceed:

```
1 TFoo = class
2   loc_a: Integer;           // Find references on loc_a finds only
3   procedure Foo1;           // this line (Line 2) and the usage
4 end;                        // in TFoo.Foo1 (Line 15)

5 var
6  loc_a: string;           // Find references on loc_a here
                            // finds only this line (Line 6) and
                            // the usage in procedure Foo (Line11)
7 implementation

8 {$R *.nfm}

9 procedure Foo;
10 begin
11  loc_a := 'test';
12 end;

13 procedure TFoo.Foo1;
14 begin
15   loc_a:=1;
16 end;
```

See Also

Refactoring Overview (see page 57)

Refactoring Code (see page 143)

1.5.13 Change Parameters Overview (Delphi)

Adding or removing a parameter from a function is a commonly performed and tedious programming task. RAD Studio provides the **Change Parameters** refactoring to automate this task. You can use **Change Parameters** to add, remove, and rearrange function parameters.

To use this refactoring, select a function name in the **Code Editor** and choose **Refactor ▶ Change Params**.

When you use the **Change Parameters** refactoring, the following function signature conflicts can occur:

- A descendant class contains an **override** for the function you are refactoring. When you refactor the function, any functions that **override** the refactored function will also be refactored.
- A descendent class contains an overloaded version of the function that has the same signature as the refactored version. When you refactor the function, the **overload** is changed to an **override**.
- A descendent class has an overridden method that matches the original signature. When you refactor the function, the **override** is changed to an **overload**.

Note: If you remove a parameter, you need to manually remove any method code that uses the removed parameter.

See Also

Refactoring Overview (see page 57)

1.5.14 Extract Resource String

Refactor ▶ Extract Resource String

Use this dialog box to convert the string currently selected in the **Code Editor** to a resource string. The `resourcestring` keyword and the resource string will be added to the implementation section of your code, and the original string will be replaced with the new resource string name.

Item	Description
String	Displays the name of string to be extracted as a resource string.
Name	Displays a suggested string name. You can change it as needed.

See Also

Refactoring Overview (see page 57)

Isolating Resources

1.5.15 Find Unit

Refactor ▶ Find Unit

Use this dialog to locate units and add them to the uses clause of your Delphi code file.

Item	Description
Search	Displays the selected search identifier. You can change the identifier in this text box. The search automatically restricts the results based on what you type.
Matching Results	Displays all of the potentially appropriate units. Select or multiselect units in the list to add the unit references to your uses clause.
Add to the Interface	Select this to add the unit reference to the interface section.
Add to the Implementation	Select this to add the unit reference to the implementation section.

1.5.16 Sync Edit Mode (Delphi, C#, C++)

Sync Edit mode allows you to change all occurrences of an identifier when you change one instance of that identifier. When you enter Sync Edit mode, you can tab to each highlighted identifier in your current **Code Editor** window. If you change an identifier that appears elsewhere in the file, all occurrences transform to whatever you type, character by character.

See Also

Refactoring Overview (see page 57)

Using Sync Edit (see page 150)

Refactoring Code (see page 143)

1.5.17 Refactorings

View ▶ Refactorings

Performs the listed refactorings.

Item	Description
Refactor	Performs the refactoring operation on the entries.
Undo Refactoring	Undoes any refactorings that were performed immediately prior to selecting this command.
Remove Refactoring	Removes selected references from the list.
Remove Refactorings	Removes all refactorings from the list.

See Also

Refactoring Overview (see page 57)

1.5.18 Undoing a Refactoring (Delphi, C#)

The refactoring engine takes advantage of a versioning mechanism, known as *local striping*, to allow you to undo renames in source code files. The IDE records the current timestamp of each file included in the current refactoring changeset. The timestamp corresponds to a specific local revision of the file. When you select the undo command, the IDE copies the local backup file that matches that timestamp back over the refactored file.

The important point to understand is that any changes that you make to the files after the refactoring will also be rolled back when you perform an Undo. Before the Undo is applied, you will get a warning message confirming that you want to apply the Undo. Applying the Undo reverts changes back to before the refactoring was originally applied in all modified files. You will lose any changes made in those files since the refactoring was originally applied.

Undo performs local striping only for Rename because Rename is the only refactoring operation that affects multiple files.

If you want to undo Extract Method, Declare Field, or Declare Variable refactorings, use Ctrl-z (regular Undo) in the **Code Editor**, or the **Undo** button in the **Refactoring** window, which accomplishes the same thing.

See Also

Refactoring Overview (↗ see page 57)

Refactoring Code (↗ see page 143)

1.5.19 Rename Symbol (C++)

Refactor ▶Rename

Use this dialog box to specify a new name for the selected symbol before refactoring your code.

Item	Description
Old name	Displays the current symbol name.
New name	Specifies the new name for the symbol.
View references before refactoring	Displays the **Refactorings** dialog box, enabling you to preview the proposed changes and selectively rename the symbol. If this option is unchecked, the rename refactoring is performed immediately.

1.5.20 Rename <symbol name> (C#)

Refactor ▶Rename <symbol name>

Use this dialog box to perform rename refactoring on a symbol, such as a variable, type, field, method, or parameter, currently selected in the **Code Editor**. The first field in the dialog varies based on the type of symbol you are renaming.

Item	Description
Namespace	Displayed when renaming a type. Indicates the namespace in which the type is defined.
Procedure	Displayed when renaming a variable. Indicates the procedure in which the variable is selected.
Class	Displayed when renaming a field, method, or parameter. Indicates the class in which the field, method, or parameter is defined.
Old name	The current name of the selected symbol.
New name	Enter the new name for the selected symbol.
View references before refactoring	Displays the **Refactorings** dialog, enabling you to preview the proposed changes and selectively rename the symbol. If this option is unchecked, the rename refactoring is performed immediately.

Tip: The Refactor->Rename <symbol name>

command is also available from the **Code Editor** context menu.

See Also

Refactoring Overview (see page 57)

1.5.21 Rename <symbol name> (Delphi)

Refactor ▶ Rename <symbol name>

Use this dialog box to perform rename refactoring on a symbol, such as a variable, type, field, method, or parameter, currently selected in the **Code Editor**. The first field in the dialog varies based on the type of symbol you are renaming.

Item	Description
Unit	Displayed when renaming a type. Indicates the unit in which the type is defined.
Procedure	Displayed when renaming a variable. Indicates the procedure in which the variable is selected.
Class	Displayed when renaming a field, method, or parameter. Indicates the class in which the field, method, or parameter is defined.
Old name	The current name of the selected symbol.
New name	Enter the new name for the selected symbol.
View references before refactoring	Displays the **Refactorings** dialog box, enabling you to preview the proposed changes and selectively rename the symbol. If this option is unchecked, the rename refactoring is performed immediately.

Tip: The Refactor->Rename <symbol name>

command is also available from the **Code Editor** context menu.

See Also

Refactoring Overview (see page 57)

1.6 Testing Applications

Unit testing is an integral part of developing reliable applications. The following topics discuss unit testing features included in RAD Studio.

Topics

Name	Description
Unit Testing Overview (↗ see page 70)	RAD Studio integrates two open-source testing frameworks, *DUnit* and *NUnit*, for developing and running automated test cases for your applications. The DUnit framework is available for Delphi and C++. The NUnit framework is available for Delphi for .NET and C# only. These frameworks simplify the process of developing tests for classes and methods in your application. Using unit testing in combination with refactoring can improve your application stability. Testing a standard set of tests every time a small change is made throughout the code makes it more likely that you will catch any problems early in the development cycle.... more (↗ see page 70)
DUnit Overview (↗ see page 72)	DUnit is an open-source unit test framework based on the JUnit test framework. The DUnit framework enables you to build and execute tests against Delphi Win32 applications. The RAD Studio integration of DUnit framework enables you to develop and execute tests against Delphi Win32, Delphi .NET, and C++Builder applications. Each testing framework provides its own set of methods for testing conditions. The methods represent common assertions. You can also create your own custom assertions. You can use the provided methods to test a large number of conditions.
NUnit Overview (↗ see page 76)	NUnit is an open-source unit test framework based on the JUnit test framework. The NUnit framework allows you to develop and execute tests against .NET Framework applications. The RAD Studio integration of NUnit allows you to develop and execute tests for both Delphi for .NET and C# applications. The NUnit framework is not supported in C++Builder or Delphi for Win32. This topic includes information about: • Developing NUnit Tests. • NUnit Asserts. • NUnit Test Runners.

1.6.1 Unit Testing Overview

RAD Studio integrates two open-source testing frameworks, *DUnit* and *NUnit*, for developing and running automated test cases for your applications. The DUnit framework is available for Delphi and C++. The NUnit framework is available for Delphi for .NET and C# only. These frameworks simplify the process of developing tests for classes and methods in your application. Using unit testing in combination with refactoring can improve your application stability. Testing a standard set of tests every time a small change is made throughout the code makes it more likely that you will catch any problems early in the development cycle.

Both testing frameworks are based on the JUnit test framework and share much of the same functionality.

This topic includes the following information:

- What Is Installed
- Test Projects
- Test Cases
- Test Fixtures

What Is Installed

By default, both frameworks are installed during the complete RAD Studio installation. When installing individual personalities, the test framework(s) supported by those personalities will be installed.

DUnit

For Delphi and C++Builder, the DUnit framework is installed automatically by the RAD Studio installer. You can find many DUnit resources in the `\source\DUnit` directory, under your installation root directory. These resources include source files, documentation, and test examples. For C++Builder, the following C++ header and library files are also provided for use as C++ test projects:

- GUITestRunner.hpp
- XMLTestRunner.hpp
- TextTestRunner.hpp
- TestFramework.hpp
- DUnitMainForm.hpp
- DUnitAbout.hppdir
- dunitrtl.lib

  Note: These files are not part of the standard DUnit distribution. These files are prebuilt by CodeGear and included with C++Builder for your convenience.

  In general when using DUnit, include at least one test case and one or more test fixtures. Test cases typically include one or more assertion statements to verify the functionality of the class being tested.

  DUnit is licensed under the Mozilla Public License 1.0 (MPL).

NUnit

The NUnit framework is available for the Delphi for .NET and C# personalities only.

During the installation process, you are prompted to install NUnit, which you can accept or decline. You can change the location for installing NUnit, or you can accept the default; the default is `C:\Program Files\NUnit V2.x`, where *x* is a point release number. The installation directory includes a number of resources including documentation and example tests.

NUnit is the name of the .NET testing framework and can be used with both Delphi for .NET and C# projects. There are some subtle but important differences between the way NUnit and DUnit work. For example, NUnit does not link in .dcu files, but DUnit does.

In general when using NUnit, include at least one test case and one or more test fixtures. Test cases typically include one or more assertion statements to verify the functionality of the class being tested.

Test Projects

A test project encapsulates one or more test cases and is represented by a node in the IDE **Project Manager**. RAD Studio provides the **Test Project Wizard**, which you can use to create a basic test project. Once you have a test project that is associated with a code project, you can create test cases and add them to the test project.

Test Cases

In a typical unit test project, each class to be tested has a corresponding test class; however, this is not required. The test class is also referred to as a *test case*. Depending on which framework you are using, the test class may be derived from a specific test case base class. In general, a test case has a set of one or more methods that correspond to one of the methods in the class to be tested. More than one test case can be included in a test project. This ability to group and combine tests into test cases—and test cases into test projects—is what sets a test case apart from simpler forms of testing (such as using print statements or evaluating debugger expressions). Each test case and test project can be reused and rerun, and can be automated through the use of batch files, build scripts, or other types of testing systems.

Generally, it is recommended that you create your tests in a project separate from the source file project. That way, you do not have to go through the process of removing your tests from your production application. RAD Studio provides the **Test Case Wizard** to help you create basic test cases, which you can then customize as needed.

Test Fixtures

The term *test fixture* refers to the combination of multiple test cases, which test logically related functionality. You can define test fixtures in your test case. Typically, you will instantiate your objects, initialize variables, set up database connection, and perform maintenance tasks in the `SetUp` and `TearDown` sections. As long as your tests all act upon the same objects, you can include a number of tests in any given test fixture.

See Also

DUnit Overview (see page 72)

NUnit Overview (see page 76)

Developing Tests (see page 180)

Mozilla Public License 1.0

zlib/libpng License

1.6.2 DUnit Overview

DUnit is an open-source unit test framework based on the JUnit test framework. The DUnit framework enables you to build and execute tests against Delphi Win32 applications. The RAD Studio integration of DUnit framework enables you to develop and execute tests against Delphi Win32, Delphi .NET, and C++Builder applications.

Each testing framework provides its own set of methods for testing conditions. The methods represent common assertions. You can also create your own custom assertions. You can use the provided methods to test a large number of conditions.

Developing Delphi DUnit Tests

Every DUnit test implements a class of type `TTestCase`.

The following sample Delphi Win32 program defines two functions that perform simple addition and subtraction:

```
unit CalcUnit;

interface

type

{ TCalc }

  TCalc = class
  public
    function Add(x, y: Integer): Integer;
    function Sub(x, y: Integer): Integer;
  end;

implementation

{ TCalc }

function TCalc.Add(x, y: Integer): Integer;
begin
  Result := x + y;
end;
```

```
function TCalc.Sub(X, Y: Integer): Integer;
begin
  Result := x + y;
end;

end.
```

The following example shows the test case skeleton file that you need to modify to test the two functions, `Add` and `Sub`, in the preceding code.

```
unit TestCalcUnit;

interface

uses
  TestFramework, CalcUnit;
type
  // Test methods for class TCalc
  TestTCalc = class(TTestCase)
  strict private
    aTCalc: TCalc;
  public
    procedure SetUp; override;
    procedure TearDown; override;
  published
    procedure TestAdd;
    procedure TestSub;
  end;

implementation

procedure TestTCalc.SetUp;
begin
  aTCalc := TCalc.Create;
end;

procedure TestTCalc.TearDown;
begin
  aTCalc := nil;
end;

procedure TestTCalc.TestAdd;
var
  _result: System.Integer;
  y: System.Integer;
  x: System.Integer;
begin
  _result := aTCalc.Add(x, y);
  // TODO: Add testcode here
end;

procedure TestTCalc.TestSub;
var
  _result: System.Integer;
  y: System.Integer;
  x: System.Integer;
begin
  _result := aTCalc.Sub(x, y);
  // TODO: Add testcode here
end;

initialization
  // Register any test cases with the test runner
  RegisterTest(TestTCalc.Suite);
end.
```

Developing C++ DUnit Tests

Every DUnit test implements a class of type TTestCase.

The following sample C++ Win32 header file and program define two functions that perform simple addition and subtraction:

```
#ifndef Unit7H
#define Unit7H
//-----------------------------------------------------


class TCalc
{
public:
  int Add(int x, int y);
  int Sub(int x, int y);
};


#endif
```

The following example (TestUnit7.cpp) contains a Testcase for the TCalc class. The Wizard generated this example, but the user is expected to write tests that exercise the functions Add and Sub. The example illustrates the DUnit scaffolding for Unit Tests.

```
#include <vcl.h>
#pragma hdrstop

#include <TestFramework.hpp>

class TTestTCalc : public TTestCase
{
public:
  __fastcall virtual TTestTCalc(AnsiString name) : TTestCase(name) {}
  virtual void __fastcall SetUp();
  virtual void __fastcall TearDown();

__published:
  void __fastcall TestAdd();
  void __fastcall TestSub();
};


void __fastcall TTestTCalc::SetUp()
{
}

void __fastcall TTestTCalc::TearDown()
{
}

void __fastcall TTestTCalc::TestAdd()
{
  // int Add(int x, int y)
}

void __fastcall TTestTCalc::TestSub()
{
  // int Sub(int x, int y)
}



static void registerTests()
{
  _di_ITestSuite iSuite;
  TTestSuite* testSuite = new TTestSuite("Testing Unit7.h");
```

```
    if (testSuite->GetInterface(iSuite)) {
      testSuite->AddTests(__classid(TTestTCalc));
      Testframework::RegisterTest(iSuite);
    } else {
      delete testSuite;
    }
  }
  #pragma startup registerTests 33


  /* [For debug purposes only - To be removed soon!!]
  GenerateHeaderComment=true
  DefaultExtension=.cpp
  FileName=C:\Users\bbabet\Documents\RAD Studio\Projects\DUnitSample\Test\TestUnit7.cpp
  TestFramework=DUnit / C++ Win32
  OutputPersonality=CPlusPlusBuilder.Personality
  TestProject=C:\Users\bbabet\Documents\RAD Studio\Projects\DUnitSample\Test\Project3Tests.cbproj
  UnitUnderTest=C:\Users\bbabet\Documents\RAD Studio\Projects\DUnitSample\Unit7.h
  NameOfUnitUnderTest=Unit7.h
  TestCaseBaseClass=TTestCase
  TestCasePrefix=Test
  UnitName=TestUnit7
  Namespace=TestUnit7
  TestClasses=
    <0>
      Name=TCalc
      Methods=
        <0>
          Name=Add
          Signature=int Add(int x, int y)
        <1>
          Name=Sub
          Signature=int Sub(int x, int y)
  TestClass=
  Method=
  */
```

DUnit Functions

DUnit provides a number of functions that you can use in your tests.

Function	Description
Check	Checks to see if a condition was met.
CheckEquals	Checks to see that two items are equal.
CheckNotEquals	Checks to see if items are not equal.
CheckNotNull	Checks to see that an item is not null.
CheckNull	Checks to see that an item is null.
CheckSame	Checks to see that two items have the same value.
EqualsErrorMessage	Checks to see that an error message emitted by the application matches a specified error message.
Fail	Checks that a routine fails.
FailEquals	Checks to see that a failure equals a specified failure condition.
FailNotEquals	Checks to see that a failure condition does not equal a specified failure condition.
FailNotSame	Checks to see that two failure conditions are not the same.
NotEqualsErrorMessage	Checks to see that two error messages are not the same.
NotSameErrorMessage	Checks that one error message does not match a specified error message.

For more information on the syntax and usage of these and other DUnit functions, see the DUnit help files in

`\source\dunit\doc`.

DUnit Test Runners

A test runner allows you to run your tests independently of your application. In a DUnit test project, the test runner code from the DUnit framework is compiled directly into the generated executable making the test project itself a test runner. This is different from the NUnit framework, which uses a separate test runner executable for running tests. The integrated DUnit framework provides two test runners:

- *The GUI Test Runner* — This displays test results interactively in a GUI window, with results color-coded to indicate success or failure.
- *The Console Test Runner* — This sends all test output to the console.

The DUnit GUI Test Runner is very useful when actively developing unit tests for the code you are testing. The DUnit GUI Test Runner displays a green bar over a test that completes successfully, a red bar over a test that fails, and a yellow bar over a test that is skipped.

The DUnit Console Test Runner is useful when you want to run completed code and tests from automated build scripts.

See Also

Unit Testing Overview (see page 70)

NUnit Overview (see page 76)

Developing Tests (see page 180)

1.6.3 NUnit Overview

NUnit is an open-source unit test framework based on the JUnit test framework. The NUnit framework allows you to develop and execute tests against .NET Framework applications. The RAD Studio integration of NUnit allows you to develop and execute tests for both Delphi for .NET and C# applications. The NUnit framework is not supported in C++Builder or Delphi for Win32.

This topic includes information about:

- Developing NUnit Tests.
- NUnit Asserts.
- NUnit Test Runners.

Developing NUnit Tests

Each testing framework provides its own set of methods for testing conditions. The methods support common assertions. You can also create your own custom assertions. You will be able to use the provided methods to test a large number of conditions.

If you want to create tests for an application, you first create a *test project*. The test project contains the *test case* files, which contain one or more tests. A test case is analogous to a class. Each test is analogous to a method. Typically, you might create one test for each method in your application. You can test each method in your application classes to make sure that the method performs the task you expect.

You can use the **Test Project Wizard** to generate a test project file, which builds a .NET assembly that must be run by one of the NUnit test runners. You can then use the **Test Case Wizard** to generates a skeleton test method for each method in the class being tested. This includes local variable declarations for each of the parameters to the method being called. You will need to write the code required to set up the parameters for the call, and the code to verify the return values or other state that is appropriate following the call.

The following example shows a small C# program that performs simple addition and subtraction:

```
using System;
```

```
namespace CSharpCalcLib
{
    /// <summary>
    /// Simple Calculator Library
    /// </summary>
    public class Calc
    {
        public int Add(int x, int y)
        {
            return x + y;
        }

        public int Sub(int x, int y)
        {
            return x + y;
        }
    }
}
```

The following example shows the test case skeleton file that you need to modify to test the two methods, Add and Sub, in the preceding code.

```
namespace TestCalc
{
    using System;
    using System.Collections;
    using System.ComponentModel;
    using System.Data;
    using NUnit.Framework;
using CSharpCalcLib;


// Test methods for class Calc
[TestFixture]
public class TestCalc
{

private Calc aCalc;

[SetUp]
public void SetUp()
{
aCalc = new Calc();
}

[TearDown]
public void TearDown()
{
aCalc = null;
}

[Test]
public void TestAdd()
{
int x;
int y;
int returnValue;
// TODO: Setup call parameters
returnValue = aCalc.Add(x, y);
// TODO: Validate return value
}

[Test]
public void TestSub()
{
```

```
int x;
int y;
int returnValue;
// TODO: Setup call parameters
returnValue = aCalc.Sub(x, y);
// TODO: Validate return value
}
}
}
```

Note: Each test method is automatically decorated with the [Test]

attribute in C# projects. In addition, in C# the test methods are defined as functions returning void. The following example shows a small Delphi for .NET program that performs simple addition and subtraction:

```
unit CalcUnit;

// .Net Version

interface

type

{ TCalc }

  TCalc = class
  public
    function Add(x, y: Integer): Integer;
    function Sub(x, y: Integer): Integer;
  end;

implementation

{ TCalc }

function TCalc.Add(x, y: Integer): Integer;
begin
  Result := x + y;
end;

function TCalc.Sub(X, Y: Integer): Integer;
begin
  Result := x + y;
end;

end.
```

The following example shows the test case skeleton file that you need to modify to test the two functions, Add and Sub, in the preceding code.

```
unit TestCalcUnit;

interface

uses
  NUnit.Framework, CalcUnit;

type
  // Test methods for class TCalc
  [TestFixture]
  TestTCalc = class
  strict private
    FCalc: TCalc;
  public
    [SetUp]
    procedure SetUp;
    [TearDown]
    procedure TearDown;
  published
    [Test]
    procedure TestAdd;
    [Test]
    procedure TestSub;
  end;

implementation

procedure TestTCalc.SetUp;
begin
  FCalc := TCalc.Create;
end;

procedure TestTCalc.TearDown;
begin
  FCalc := nil;
end;

procedure TestTCalc.TestAdd;
var
  ReturnValue: Integer;
  y: Integer;
  x: Integer;
begin
  // TODO: Setup call parameters
  ReturnValue := FCalc.Add(x, y);
  // TODO: Validate return value
end;

procedure TestTCalc.TestSub;
var
  ReturnValue: Integer;
  y: Integer;
  x: Integer;
begin
  // TODO: Setup call parameters
  ReturnValue := FCalc.Sub(x, y);
  // TODO: Validate return value
end;

end.
```

Note: In Delphi for .NET the test methods are defined as procedures.

Each test method must:

- be public
- be a procedure for Delphi for .NET or a function with a void return type for C#
- take no arguments

Setup

Use the `SetUp` procedure to initialize variables or otherwise prepare your tests prior to running. For example, this is where you would set up a database connection, if needed by the test.

TearDown

The `TearDown` method can be used to clean up variable assignments, clear memory, or perform other maintenance tasks on your tests. For example, this is where you would close a database connection.

NUnit Asserts

NUnit provides a number of asserts that you can use in your tests.

Function	Description	Syntax
AreEqual	Checks to see that two items are equal.	`Assert.AreEqual(expected, actual [, string message])`
IsNull	Checks to see that an item is null.	`Assert.IsNull(object [, string message])`
IsNotNull	Checks to see that an item is not null.	`Assert.IsNotNull(object [, string message])`
AreSame	Checks to see that two items are the same.	`Assert.AreSame(expected, actual [, string message])`
IsTrue	Checks to see that an item is **True**.	`Assert.IsTrue(bool condition [, string message])`
IsFalse	Checks to see that an item is **False**.	`Assert.IsFalse(bool condition [, string message])`
Fail	Fails the test.	`Assert.Fail([string message])`

You can use multiple asserts in any test method. This collection of asserts should test the common functionality of a given method. If an assert fails, the entire test method fails and any other assertions in the method are ignored. Once you fix the failing test and rerun your tests, the other assertions will be executed, unless one of them fails.

NUnit Test Runners

A test runner allows you to run your tests independently of your application. The NUnit test runners work by loading an assembly that contains the unit tests. The test runners identify the tests in the assembly from the `[TEST]` attributes generated by the **Test Case Wizard**.

NUnit includes two test runner executables:

- `NUnitConsole.exe` — This is the *Console Test Runner*, a text-based test runner that sends test output to the console.
- `NUnitGUI.exe` — This is the *GUI Test Runner*, an interactive GUI-based test runner, with results color-coded for success or failure.

The GUI Test Runner is very useful when actively developing unit tests or the code you are testing. The GUI Test Runner displays a green indicator for a test that completes successfully, a red indicator for a test that fails, and a yellow indicator for a test that is skipped.

The Console Test Runner is useful when you need to run completed code and tests from automated build scripts. If you want to redirect the output to a file, use the redirection command parameter. The following example shows how to redirect test results to a `TestResult.txt` text file:

```
nunit-console nunit.tests.dll /out:TestResult.txt
```

Note: You may need to set the path to your host application in the Project Options

dialog. Set the Host Application property to the location of the test runner you want to use.

See Also

Unit Testing Overview (↗ see page 70)

DUnit Overview (↗ see page 72)

Building Test Cases (↗ see page 180)

NUnit.org Documentation

1.7 Modeling Applications with Together

This section provides an overview of the features provided by Together.

Note: The product version you have determines which Together features are available.

Topics

Name	Description
Getting Started with Together (↗ see page 83)	The two sample projects are designed to help you explore Together features while working with projects. Some of the special features include: UML modeling, patterns, generating project documentation.
Modeling Overview (↗ see page 89)	Effective modeling with Together simplifies the development stage of your **project**. Smooth integration to RAD Studio provides developers with easy transition from **models** to source code. The primary objective of modeling is to organize and visualize the structure and components of software intensive systems. **Models** visually represent requirements, subsystems, logical and physical elements, and structural and behavioral patterns. While contemporary software practices stress the importance of developing models, Together extends the benefits inherent to modeling by fully synchronizing **diagrams** and source code.
Together Project Overview (↗ see page 89)	Work in Together is done in the context of a **project**. A project is a logical structure that holds all resources required for your work. Together works with the following project types: **design** and **implementation**. Each of them includes several project formats. It is up to you to define which directories, archives, and files should be included in your project. You can set up project properties when the project is being created, and modify them further, using the Object InspectorProperties Window.
Namespace and Package Overview (↗ see page 89)	A **namespace** is an element in a model that contains a set of named elements that can be identified by name. A project consists of one or more namespaces (or packages). A namespace and a package are almost synonyms: the term “namespace” is used for implementation projects, the term “package” is used for design projects. A namespace (or a package) is like a box where you put diagrams and model elements. Contents of a namespace (package) can be displayed on a special type of the Class Diagram. Each project contains the **default** namespace (or package) just after its creation.
Together Diagram Overview (↗ see page 90)	**Diagrams** can be thought of as graphs with vertices and edges that are arranged according to a certain **algorithm**. Each diagram belongs to a certain **diagram type** (for example, UML 2.0 Class Diagram). A set of **model elements** available for use on a diagram depends on the diagram type. Diagrams exist within the context of a namespace (or a package). You have to create or open a project or project groupsolution before creating a new diagram. When Together support is activated, the project-level package diagram is created by default. You can create the various UML diagrams in the... more (↗ see page 90)
Supported UML Specifications (↗ see page 90)	The Object Management Group’s Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of distributed object systems. Together supports UML to help you specify, visualize, and document models of your software systems, including their structure and design. Refer to UML documentation for the detailed information about UML semantics and notation. The *UML (version): Superstructure* document defines the user level constructs required for UML. It is complemented by the *UML (version): Infrastructure* document which defines the foundational language constructs required for UML. The two complementary specifications constitute a complete specification for the UML modeling... more (↗ see page 90)
Model Element Overview (↗ see page 91)	**Model element** is any component of your model that you can put on a diagram. Model elements include **nodes** and **links** between them. A set of available model elements depends on a current diagram type. Available model elements are displayed in the Tool PaletteToolbox. A link can have a label. You can move a label to any point of the link line.

Model Annotation Overview (see page 91)	The Tool PaletteToolbox for UML diagram elements displays note and note link buttons for all UML diagrams. Use these elements to place annotation nodes and their links on the diagram. Notes can be free floating or you can draw a note link to some other element to show that a note pertains specifically to it. You can attach a note link to another link. The text of notes linked to class diagram elements does not appear in the source code.
Model Shortcut Overview (see page 92)	A **shortcut** is a representation of an existing node element placed on the same or a different diagram. Shortcuts facilitate reuse of elements, make it possible to display library classes on diagrams, and demonstrate relationships between the diagrams within the model. You can create a shortcut to an element of any other project in the current project groupsolution. You can create a shortcut to an inner class or interface of another classifier. It is also possible to add a shortcut to an element from project **References**, including binary (`.dll`, `.exe`) files. The small special... more (see page 92)
Diagram Layout Overview (see page 92)	You can customize arrangement of model elements on your diagrams automatically or manually. Together enables you to manage diagrams with automated layout features that optimize the diagram layout for viewing or printing. Nodes and links on a diagram are arranged according to a certain algorithm. There are several diagram layout algorithms available. You can apply the same algorithm for the entire model, or different algorithms to separate diagrams. Each algorithm has a set of specific options defined in the **Together ▶ (level) ▶ Diagram ▶ Layout** category of the **Options** dialog window. It is also possible to lay out a diagram manually by... more (see page 92)
Model Hyperlinking Overview (see page 92)	You can create hyperlinks from diagrams or diagram elements to other system artifacts and browse directly to them.
LiveSource Overview (see page 93)	LiveSource™ is the key feature of Together that keeps your model and source code in sync. That is why it applies to implementation projects only. When a Class diagram is created in an implementation project, it is immediately synchronized with the implementation code. When you change a Class diagram, Together updates the corresponding source code. Together allows you to synchronize different aspects of your project in several ways. Use the **Reload** command to refresh the Together model from the source code.
Transformation to Source Code Overview (see page 94)	Together enables you to generate source code based on a language-neutral design project.
OCL Support Overview (see page 95)	
Patterns Overview (see page 96)	**Patterns** provide software developers with powerful reuse facilities. Rather than trying to tackle each design problem from the very outset, you can use the predefined patterns supplied with Together. The hierarchy of patterns is defined in the **Pattern Registry**. You can manage and logically arrange your patterns using the **Pattern Organizer**. Patterns are intended to: • Create frequently used elements • Modify existing elements • Implement useful source code constructions or project groupsolutions in your model
Refactoring Overview (see page 98)	Together provides extensive support for refactoring your implementation projects. Refactoring means rewriting existing source code with the intent of improving its design rather than changing its external behavior. The focus of refactoring is on the structure of the source code, changing the design to make the code easier to understand, maintain, and modify. The refactoring features provided by Together affect both source code and model. As a result, your project is consisting after refactoring, even if it includes UML diagrams. The primary resource book on refactoring is *Refactoring - Improving the Design of Existing Code by Martin Fowler (Addison -... more (see page 98)*

Quality Assurance Facilities Overview (↗ see page 98)	Together provides audits and metrics as Quality Assurance features to unobtrusively help you enforce company standards and conventions, capture real metrics, and improve what you do. Although audits and metrics are similar in that they both analyze your code, they serve different purposes. Audits and metrics are run as separate processes. Because the results of these two processes are different in nature,Together provides different features for interpreting and organizing the results. Note that some of the features and procedures described in this section apply to both audits and metrics while some are specific to one or the other.
Documentation Generation Facility Overview (↗ see page 100)	This feature automatically generates documentation for your project. Use this feature to illustrate you programme with the documentation in the HTML format. You can update this automatically generated documentation when your project changes, or edit this documentation manually afterwards.
Model Import and Export Overview (↗ see page 100)	You can share model information with other systems by importing and exporting model information, or by sharing project files: ***Import and export features***

1.7.1 Getting Started with Together

The two sample projects are designed to help you explore Together features while working with projects. Some of the special features include: UML modeling, patterns, generating project documentation.

Topics

Name	**Description**
About Together (↗ see page 83)	Welcome to **CodeGear® Together®**, the award-winning, design-driven environment for modeling applications. Together includes features such as support for UML 2.0, OCL, patterns, Quality Assurance audits and metrics, source code refactoring and generation, IBM Rational Rose (MDL) format import, XMI format import and export, and automated documentation generation. A key feature of Together, LiveSource™, keeps your Together diagrams synchronized with your source code in the RAD Studio Editor. The Together features are tightly integrated with the RAD Studio environment. When Together support is activated, the following items are added or modified: • **Diagram View** • **Model View** • Object InspectorProperties Window • Tool... more (↗ see page 83)
UML 2.0 Sample Project (↗ see page 84)	This section contains a UML 2.0 sample project guide. **Important**: This sample project is available for RAD Studio Architect version only. Follow the steps:
Code Visualization Overview (↗ see page 87)	The Code Visualization feature is available in both the Enterprise and Architect versions of RAD Studio. All other modeling tools and information related to modeling relates only to the Architect version of RAD Studio.
What's New in Together (↗ see page 88)	This version includes the following new and improved features. • Support for Microsoft VS .NET 2005 (instead of 2003) • .NET 2.0 support for C# .NET and VB .NET • Structured XML documentation technology

1.7.1.1 About Together

Welcome to **CodeGear® Together®**, the award-winning, design-driven environment for modeling applications. Together includes features such as support for UML 2.0, OCL, patterns, Quality Assurance audits and metrics, source code refactoring and generation, IBM Rational Rose (MDL) format import, XMI format import and export, and automated documentation generation.

A key feature of Together, LiveSource™, keeps your Together diagrams synchronized with your source code in the RAD Studio Editor.

The Together features are tightly integrated with the RAD Studio environment. When Together support is activated, the following items are added or modified:

- **Diagram View**
- **Model View**
- Object InspectorProperties Window
- Tool PaletteToolbox

In addition, specific commands are added to the main menu and the context menus of the Project ManagerSolution Explorer and Structure ViewClass View.

Warning: Not all features described in this Help are available in all editions of the product.

If your Internet access is limited by network security, or if your computer is protected by a personal firewall, the Web-based links in this Help system might not function properly.

See Also

Modeling Overview (see page 89)

Help on Help (see page 51)

Together Documentation Set

Together Glossary (see page 1139)

Keyboard Shortcuts (see page 1104)

1.7.1.2 UML 2.0 Sample Project

This section contains a UML 2.0 sample project guide.

Important: This sample project is available for RAD Studio Architect version only.

Follow the steps:

Topics

Name	Description
UML 2.0 Sample Project, Behavior Package (see page 85)	This package contains diagrams: • Activity diagram • State Machine diagram • Use Case diagram • Interaction diagram
UML 2.0 Sample Project, Structure Package (see page 86)	The default diagram on the top level of the project contains two packages intended to demonstrate the structural and behavioral modeling. This package contains the following diagrams: • Class diagram • Component diagram • Composite Structure diagram • Deployment diagram

1.7.1.2.1 UML 2.0 Sample Project, Behavior Package

This package contains diagrams:

- Activity diagram
- State Machine diagram
- Use Case diagram
- Interaction diagram

Activity diagram

Activity diagram features are presented by three diagrams:

- Data Activity
- Final Nodes
- Process Order

Data Activity diagram

Data Activity diagram demonstrates object flow via actions and pins, and the usage of central buffer.

Final Nodes diagram

Final Nodes diagram represents a process of building an application, using multiple control flows and terminal blocks. During the process the application components should be assembled. If there are no more components to be built, the building flow is terminated (Flow Final), while the installation flow goes on working. If all the components are built and installed, the Deliver action is performed, and thus the whole activity is terminated (Activity Final).

Process Order diagram

Process Order demonstrates the usage of interaction of the various actions by means of the control flows, transferring information via object flows, and the usage of signal send and receive elements.

State Machine diagram

State Machine diagram features are presented by two diagrams:

- Course Attempt
- Submachine State.

Course Attempt diagram

Course Attempt demonstrates the usage of substates and regions. Since the states cannot have children of the same type, the nested substates are inserted into the regions.

Submachine State diagram

Submachine State shows how one can refer to a different diagram from a state. In the case ReadAmountSM is a standalone state that represents a whole diagram, and ReadAmount:ReadAmountSM is a state that implements the behavior of ReadAmountSM. Both states are hyperlinked.

Use Case diagram

Use Case diagram is represented by Main Use Cases, which demonstrates the usage of subjects and stereotypes links.

Interaction diagrams

Interaction diagrams show interaction between objects represented by their lifelines, by ways of messages. Each lifeline instantiates a class or represents a part. An interaction can be shown in two ways: as a sequence diagram or as a communication diagram. The sample interaction ShowAlbumsDialog is represented by the ShowAlbumsDialog diagram (sequence) and cd_ShowAlbumsDialog diagram (communication).

See Also

UML 2.0 Sample Project (see page 84)

UML 2.0 Sample Project (see page 86)

Modeling overview (see page 89)

OpenUML20Sample.xml (see page 190)

1.7.1.2.2 UML 2.0 Sample Project, Structure Package

The default diagram on the top level of the project contains two packages intended to demonstrate the structural and behavioral modeling.

This package contains the following diagrams:

- Class diagram
- Component diagram
- Composite Structure diagram
- Deployment diagram

Class diagram

Class diagram is represented by three samples: Class, Classes and Associations, Classes and Features.

Class Diagram

This diagram shows the usage of patterns as first class citizens. Abstract Factory is created by a GoF pattern. Explore the specific features of adding and removing participants and pattern objects.

Classes and Associations diagram

This diagram demonstrates the following features:

- Generalization of classes
- Implementation of interfaces by classes
- Usage of association classes and n-ary associations. Note that n-ary association can only be created by means of an association class
- Usage of binary associations
- Usage of association links properties (directed and non-directed associations, link labels, client and supplier multiplicities, roles, qualifiers, constraints, and so on)

Classes and Features diagram

This diagram demonstrates the following features:

- Usage of operations and attributes in the classes and interfaces
- Usage of properties of the operations, attributes and slots (visibility, multiplicity, constraints, initial values of the attributes and

slots, return types and arguments of the operations)

- Instantiating classes by means of instance specifications
- Defining features by means of the slots

Component diagram

Component diagram is represented by the Store Components diagram, which demonstrates:

- Inner components
- Usage of the required and provided interfaces via ports
- Delegation connectors that delegate calls to a component to its subcomponents.

Composite Structure diagram

Composite Structure diagram demonstrates the usage of collaborations and parts.

Deployment diagram

Deployment diagram is represented by the Application Server diagram that demonstrates the usage of deployment specifications and artifacts.

See Also

UML 2.0 Sample Project (see page 84)

UML 2.0 Sample Project (see page 85)

Modeling overview (see page 89)

OpenUML20Sample.xml (see page 190)

1.7.1.3 Code Visualization Overview

The Code Visualization feature is available in both the Enterprise and Architect versions of RAD Studio. All other modeling tools and information related to modeling relates only to the Architect version of RAD Studio.

Code Visualization and UML Static Structure Diagrams

RAD Studio has a code visualization diagram that presents a graphical view of your source code, which is reflected directly from the code itself. When you make changes in source code, the graphical depiction on the diagram is updated automatically. The code visualization diagram corresponds to a UML *static structure* diagram. A structural view of your project focuses on UML packages, data types such as classes and interfaces, and their attributes, properties, and operations. A static structure diagram also shows the relationships that exist between these entities.

This section will explain the relationship between source code and the code visualization diagram.

Note: Code visualization, and the integrated UML modeling tools are two separate and distinct features of RAD Studio. Code visualization refers to the ability to scan an arbitrary set of source code, and map the declarations within that code onto UML notation. The resulting diagram is "live", in the sense that it always reflects the current state of the source code, but you cannot make changes directly to the code visualization diagram itself. RAD Studio's model driven UML tools go a step further, giving you the ability to design your application on the diagramming surface. The model driven tools are built on Borland Together technologies, plus the Enterprise Core Objects framework. This document covers only the code visualization diagram; please use the online Help table of contents for more information on the ECO framework.

Understanding the Relationship between Source Code and Code Visualization

RAD Studio's code visualization tools use the UML notation and conventions to graphically depict the elements declared in source code. The **Code Visualization diagram** shows you the logical relationships, or *static structure* in UML terms, of the classes, interfaces and other types defined in your project. The IDE creates the **Code Visualization diagram** by mapping certain source code constructs (such as class declarations, and implementation of interfaces) onto their UML counterparts, which are then displayed on the diagram.

Top-Level Organization: Projects and UML Packages

To begin, code visualization consists of two parts of the IDE working together: The **Model View window**, and the **Diagram View**. The **Model View window** shows you the logical structure of your projects in a tree, as opposed to the file-centric view of the **Project Manager window**. Each project in a project group is a top-level node in the **Model View tree**.

Nested within each project tree-node, you will find UML packages. Each UML package corresponds to a .NET namespace or Delphi unit declaration in your source code (.NET namespaces can span multiple source files). You can expand the UML package to reveal the types declared within.

Inheritance and Interface Implementation

The UML term for the relationship formed when one class inherits from a superclass is *generalization*. When the IDE sees an inheritance relationship in your source code, it creates a generalization link within the child class node in the **Model View tree**. On the **Diagram View**, the generalization link will be shown the using standard UML notation of a solid line with a hollow arrowhead pointing at the superclass.

The UML term for interface implementation is *realization*. Similar to the case of inheritance, the IDE creates a realization link when it sees a class declaration that implements an interface. The realization link appears within the implementor class in the **Model View tree**, and on the diagram as a dotted line with a hollow arrowhead pointing at the interface. There will be one such realization link for every interface implemented by the class.

Associations

In the UML, an *association* is a navigational link produced when one class holds a reference to another class (for example, as an attribute or property). Code visualization creates association links when one class contains an attribute or property that is a non-primitive data type. On the diagram the association link exists between the class containing the non-primitive member, and the data type of that member.

Class Members: Attributes, Operations, Properties, and Nested Types

Code visualization can also map class and interface member declarations to their UML equivalents. Within the elements on the **Code Visualization diagram**, members are grouped into four distinct categories:

- Fields: Contains field declarations. The type, and optional default value assignment are shown on the diagram.
- Methods: Contains method declarations. Visibility, scope, and return value are shown.
- Properties: Contains Delphi property declarations. The type of the property is shown.
- Classes: Contains nested class type declarations.

Standard UML syntax is used to display the UML declaration of attributes, operations, and properties. Each of the four categories can be independently expanded or collapsed to show or hide the members within.

See Also

Code Visualization Diagrams (see page 730)

Class Diagram Elements (see page 1144)

Using the Model View

1.7.1.4 What's New in Together

This version includes the following new and improved features.

- Support for Microsoft VS .NET 2005 (instead of 2003)
- .NET 2.0 support for C# .NET and VB .NET
- Structured XML documentation technology

See Also

About Together (see page 83)

Using Online Help

1.7.2 Modeling Overview

Effective modeling with Together simplifies the development stage of your **project**. Smooth integration to RAD Studio provides developers with easy transition from **models** to source code.

The primary objective of modeling is to organize and visualize the structure and components of software intensive systems. **Models** visually represent requirements, subsystems, logical and physical elements, and structural and behavioral patterns.

While contemporary software practices stress the importance of developing models, Together extends the benefits inherent to modeling by fully synchronizing **diagrams** and source code.

See Also

About Together (see page 83)

Together Project Overview (see page 89)

Together Diagram Procedures (see page 190)

1.7.3 Together Project Overview

Work in Together is done in the context of a **project**. A project is a logical structure that holds all resources required for your work. Together works with the following project types: **design** and **implementation**. Each of them includes several project formats.

It is up to you to define which directories, archives, and files should be included in your project. You can set up project properties when the project is being created, and modify them further, using the Object InspectorProperties Window.

See Also

UML 2.0 Sample Project (see page 84)

Creating a Project (see page 264)

Supported Project Formats (see page 1116)

1.7.4 Namespace and Package Overview

A **namespace** is an element in a model that contains a set of named elements that can be identified by name.

A project consists of one or more namespaces (or packages). A namespace and a package are almost synonyms: the term “namespace” is used for implementation projects, the term “package” is used for design projects.

A namespace (or a package) is like a box where you put diagrams and model elements. Contents of a namespace (package) can be displayed on a special type of the Class Diagram.

Each project contains the **default** namespace (or package) just after its creation.

See Also

Working with a Namespace or Package (see page 250)

Class Diagram Types (see page 1124)

1.7.5 Together Diagram Overview

Diagrams can be thought of as graphs with vertices and edges that are arranged according to a certain **algorithm**.

Each diagram belongs to a certain **diagram type** (for example, UML 2.0 Class Diagram). A set of **model elements** available for use on a diagram depends on the diagram type.

Diagrams exist within the context of a namespace (or a package). You have to create or open a project or project groupsolution before creating a new diagram. When Together support is activated, the project-level package diagram is created by default. You can create the various UML diagrams in the project.

In addition to the standard properties of diagrams and their elements, you can create **user properties**, represented by the Name-Value pair.

See Also

Diagram Format Overview

Creating a Diagram (see page 196)

Working with User Properties (see page 206)

1.7.6 Supported UML Specifications

The Object Management Group’s Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of distributed object systems.

Together supports UML to help you specify, visualize, and document models of your software systems, including their structure and design.

Refer to UML documentation for the detailed information about UML semantics and notation. The *UML (version): Superstructure* document defines the user level constructs required for UML. It is complemented by the *UML (version): Infrastructure* document which defines the foundational language constructs required for UML. The two complementary specifications constitute a complete specification for the UML modeling language.

UML 1.5 and UML 2.0

The set of available diagrams depends on your project type.

For design projects, both UML 1.5 and 2.0 are supported.

For implementation projects, UML 1.5 is only supported.

The version of UML is selected when a project is created. It cannot be changed later.

UML In Color

"**UML In Color**" is an optional profile to support the **modeling in color** methodology. Color modeling makes it possible to analyze a problem domain and easily spot certain classes during analysis. Together supports the use of the four main groups of the color-modeling stereotypes:

- Role
- MomentInterval, Mi-detail
- Party, Place, Thing
- Description

For each of these stereotypes you can choose a specific color to make your model more understandable at a glance. Note that the other stereotypes do not have associated colors.

See also *"Java Modeling in Color with UML: Enterprise Components and Process" by Coad, Lefebvre and De Luca*.

See Also

http://www.uml.org/

Diagram Appearance Options (see page 1089)

1.7.7 Model Element Overview

Model element is any component of your model that you can put on a diagram.

Model elements include **nodes** and **links** between them.

A set of available model elements depends on a current diagram type. Available model elements are displayed in the Tool PaletteToolbox.

A link can have a label. You can move a label to any point of the link line.

See Also

Creating a Single Model Element (see page 209)

Tool Palette (see page 1114)

1.7.8 Model Annotation Overview

The Tool PaletteToolbox for UML diagram elements displays note and note link buttons for all UML diagrams. Use these elements to place annotation nodes and their links on the diagram.

Notes can be free floating or you can draw a note link to some other element to show that a note pertains specifically to it.

You can attach a note link to another link.

The text of notes linked to class diagram elements does not appear in the source code.

See Also

Annotating a Diagram (see page 195)

1.7.9 Model Shortcut Overview

A **shortcut** is a representation of an existing node element placed on the same or a different diagram.

Shortcuts facilitate reuse of elements, make it possible to display library classes on diagrams, and demonstrate relationships between the diagrams within the model.

You can create a shortcut to an element of any other project in the current project groupsolution. You can create a shortcut to an inner class or interface of another classifier. It is also possible to add a shortcut to an element from project **References**, including binary (`.dll`, `.exe`) files.

The small special symbol appears over a node to indicate a shortcut. It appears only if this node belongs to a different namespace or package.

Select a shortcut on your diagram and choose Navigate To Element on the context menu to navigate to the source element in the **Model View**.

See Also

Creating a Shortcut (↗ see page 208)

1.7.10 Diagram Layout Overview

You can customize arrangement of model elements on your diagrams automatically or manually.

Together enables you to manage diagrams with automated layout features that optimize the diagram layout for viewing or printing. Nodes and links on a diagram are arranged according to a certain algorithm.

There are several diagram layout algorithms available. You can apply the same algorithm for the entire model, or different algorithms to separate diagrams.

Each algorithm has a set of specific options defined in the **Together ▶ (level) ▶ Diagram ▶ Layout** category of the **Options** dialog window.

It is also possible to lay out a diagram manually by moving and resizing nodes and reshaping links. You can also lay out your diagram automatically, and then adjust arrangement manually.

See Also

Laying Out a Diagram Automatically (↗ see page 202)

Diagram Layout Algorithms (↗ see page 963)

Diagram Layout Options (↗ see page 1091)

1.7.11 Model Hyperlinking Overview

You can create hyperlinks from diagrams or diagram elements to other system artifacts and browse directly to them.

Why use hyperlinking?

Use hyperlinks for the following purposes:

- Link diagrams that are generalities or overviews to specifics and details.
- Create browse sequences leading through different but related views in a specific order; create hierarchical browse sequences.
- Link descendant classes to ancestors; browse hierarchies.
- Link diagrams or elements to standards or reference documents or generated documentation.
- Facilitate collaboration among team members.

Create a hyperlink from an existing diagram or one of its elements to any other diagram or diagram element in the project, or create a new diagram that will be hyperlinked to the current diagram.

You can also create hyperlinks from your diagrams to external documents such as files or URLs. For most users, such hyperlinking will probably take the form of documents on a LAN or document server or URLs on the company intranet. However, you can also easily link to online information such as newsgroups or discussion forums. If it is available online, you can link to it.

Hyperlink types

You can create hyperlinks to:

- An existing diagram or diagram element anywhere in the project groupsolution
- A new diagram (it will be created on-the-fly)
- A document or file on a local or remote storage device
- A URL on your company intranet or the Internet

Browse-through sequence

The hyperlinking feature of Together allows you to create browse-through sequences comprised of any number of use case or any other diagrams.

By browsing the hyperlink sequence, you can follow the relationships between the use case diagrams.

Together does not confine hyperlinking to such sequences, however. You can use hyperlinking to link diagrams and elements based on your requirements. For example, you can create a hierarchical browse-through sequence of use case diagrams, creating hyperlinks within the diagrams that follow a specific actor through all use cases that reference the actor.

See Also

Hyperlinking Diagrams (↗ see page 201)

Creating a Browse-Through Sequence of Diagrams (↗ see page 246)

1.7.12 LiveSource Overview

LiveSource™ is the key feature of Together that keeps your model and source code in sync. That is why it applies to implementation projects only.

When a Class diagram is created in an implementation project, it is immediately synchronized with the implementation code. When you change a Class diagram, Together updates the corresponding source code.

Together allows you to synchronize different aspects of your project in several ways.

Use the **Reload** command to refresh the Together model from the source code.

About MDA

Together supports the OMG’s Model Driven Architecture (MDA) initiative.

MDA is an evolving conceptual architecture for a set of industry-wide technology specifications that will support a model-driven approach to software development.

MDA is supported by UML, XMI, and other technologies.

Doc comment properties

Some properties that are defined for the model elements and members in the Object InspectorProperties Window, are presented in the source code as language-specific doc comments. In particular, these properties are: author, since, version, stereotype, associates, and so on. When such comments are encountered in the source code, they are reverse engineered to model properties.

Doc comments are presented as XML tags, preceeded by `///` (for C# projects) or `'` (for Visual Basic) However, for the sake of backward compatibility with Together ControlCenter, Together recognizes doc comment properties created in TCC for the Visual Basic projects, where the legacy format `'@property value` was used..

So doing, if the properties of an element are presented in the legacy format and one of these properties is changed to the new format `'<property> value</property>`, all the other properties are also converted.

See Also

Roundtrip Engineering Overview

Synchronizing the Model View (see page 267)

LiveSource Rules (see page 1125)

Reload Command (Model View) (see page 1112)

1.7.13 Transformation to Source Code Overview

Together enables you to generate source code based on a language-neutral design project.

About transformation to source code

You can generate source code from the Class Diagrams of your UML 1.5 or 2.0 design project and add this source code to a project in one of the supported languages. The target implementation project must already exist in the same project groupsolution.

Alternatively, you can import source code from an external design project into your current implementation project.

Name mapping

You can force Together to generate different names for your model elements in the source code. For example, you can have `ClassItem` in your source code for the `Class1` element in your model.

This feature is especially useful, if your model names are not English. You can use names in Japanese and other languages on your diagrams, but keep names in Latin alphabet in your code.

If you enable this feature, the file `codegen_map.xml` is created in the model support folder of the source design project. You can edit it with any XML or text editor. This file contains a mapping table, where each entry (model element) has two names: one for the source design project (attribute `name`), and another one for the destination implementation project (attribute `alias`). There are several sections in this file: **Class**, **Attribute**, **Operation** and **Package** for UML 1.5 projects, and **Class** and **Package** for UML 2.0 projects. Attributes name must be unique for all entries in a section.

You can optionally create an XML file with the same name and structure in a folder of any package.

Then, if you transform your project to source code and the name mapping feature is enabled, Together searches for the `codegen_map.xml` file for each model element. If the file is absent for a current package, Together searches in a parent package, and so on.

Note: If you add a new element to your model later and then transform the project to source code, Together adds a new entry for this item to the corresponding `codegen_map.xml` file. The existing entries are not changed.

See Also

LiveSource Overview (see page 93)

Transforming a Design Project to Source Code (see page 269)

1.7.14 OCL Support Overview

About OCL

The Object Constraint Language (**OCL**) is a textual language, especially designed for use in the context of diagrammatic languages such as the UML. OCL was added to UML, as it turned out a visual diagram-based language is limited in its expressiveness.

OCL 2.0 is the newest version of the OMG's constraint language to accompany their suit of Object Oriented modelling languages.

The use of OCL as an accompanying constraint and query language for modelling with these languages is essential.

Note: Portions of this product include the Object Constraint Language Library, courtesy of Kent University, United Kingdom. See http://www.cs.kent.ac.uk/projects/ocl/

OCL constraint and expression

OCL constraint

The Tool PaletteToolbox on some types of diagrams (for example, UML 2.0 Class Diagram) contains buttons that enable you to create **OCL constraints** as design elements on diagrams, and link these constraints with the desired **context**.

You can show or hide constraint elements for the better presentation of your diagrams.

OCL support for constraints provides error highlighting. The text of the constraint is validated when the constraint is linked to its context. The valid constraints are displayed in the regular font; invalid constraints, or OCL expressions with syntax errors, are displayed in a red font.

Constrained elements are marked with the decorators. The decorators are small icons attached to the context elements of constraints. If a constraint is valid the decorator is green; otherwise the decorator is red. If the constraints are concealed, you can still monitor the validity of constraints by means of the decorators.

Any OCL constraint contains an **OCL expression**.

OCL expression

For OCL expressions without object constraints (expressions as properties of other nodes), no validation is performed since no valid OCL context can be set for these elements.

Supported diagram types

OCL is supported for the following diagram types:

Diagram types with OCL support

Diagram type	Version of UML	How support is provided
Class (class, namespace, package)	UML 1.5, 2.0	Creating object constraints is supported.
Interaction (Sequence and Communication)	UML 2.0	State invariant constraints for lifelines and constraints for the operands of the combined fragments as OCL expressions.
State Machine	UML 2.0	Guard conditions of transitions as OCL expressions.
Use Case	UML 2.0	Pre- and post-condition constraints for the behavior associated with the use cases as OCL expressions. For example, an interaction chosen as a behavior.

See Also

Modeling Overview (↗ see page 89)

Together Object Constraint Language (OCL) Procedures (↗ see page 248)

OCL Editor (Diagram View) (↗ see page 1106)

1.7.15 Patterns Overview

Patterns provide software developers with powerful reuse facilities. Rather than trying to tackle each design problem from the very outset, you can use the predefined patterns supplied with Together. The hierarchy of patterns is defined in the **Pattern Registry**. You can manage and logically arrange your patterns using the **Pattern Organizer**.

Patterns are intended to:

- Create frequently used elements
- Modify existing elements
- Implement useful source code constructions or project groupsolutions in your model

Pattern Registry

The **Pattern Registry** defines the virtual hierarchy of patterns. You can create virtual folders and group the patterns logically to meet your specific requirements. All operations with the contents of the **Pattern Registry** are performed in the **Pattern Organizer** and synchronized with the **Pattern Registry**.

Pattern Organizer

The **Pattern Organizer** enables you to logically organize patterns (using virtual trees, folders and shortcuts), and view and edit the pattern properties. You will be working with shortcuts, not with the actual patterns. Because of this, shortcuts to the same pattern may be included in several folders.

Templates

Together supports templates as a way to provide backward compatibility with the legacy Together ControlCenter projects. You can copy the folders with your legacy source code templates to the `Patterns` subfolder of your Together installation directory, and use these templates to create elements in implementation projects.

Templates are text files with the language-specific extensions that use macros to be substituted with real values when the templates are applied. Therefore, templates can be regarded as forms ready for "filling in" for a specific instance. A template consists of a template file containing source code, and a properties file that contains macro descriptions and their default values.

Templates are stored in the `Patterns\templates` directory of your Together installation using the following structure:

```
/<language>/<category>/<template name>
```

where `<category>` is CLASS, LINK or MEMBER. Each `<template_name>` folder contains the following files:

- `%Name%.<ext>`
- `<template_name>,properties` (optional)

Design patterns

A design pattern is an XML file that contains a sequence of statements or actions, required to create entities and links and set their properties. Each statement creates either one model element or one link between the model elements.

In addition to creating new elements, you can use design patterns to add members to a container element. The pattern that you are applying to the specified container element should have its Use Existent property set as True. You can then apply the pattern to the container element you want to modify. For example, if you want to add several methods stored in a class as pattern to an existing class, then you have to apply that pattern to the diagram where that class exists.

The design patterns are stored as XML files in the Patterns directory of your Together installation.

Patterns as First Class Citizens

A First Class Citizen (FCC) pattern is a specific type of design pattern that contains information about the pattern name and the role of each participant. When applied to a diagram, FCC patterns create their own entities and display on the diagram with links to the created entities. Such patterns enable further modification by means of adding new participants.

Patterns as First Class Citizens are represented by GoF patterns.

A pattern is displayed on a diagram as an oval with the pattern name and an expandable list of participants. Each participant is connected with the pattern oval by a link, labeled with the participant's role.

FCC patterns generate source code, but the oval FCC pattern elements do not. The entities created by patterns are stored in the diagram files.

Stub implementation pattern

When you create an inheritance link between a class and another abstract class or interface, the methods and members are not automatically added to the child class. This problem is solved using the Stub implementation pattern. You can also create an implementation link and stub implementation in one step by using the Implementation link and stub pattern.

If the destination of a link is an interface, the pattern makes the class-source implement that interface, and creates in a class the stubs for all of the methods found in the interface and all of its parent interfaces.

If the destination link is an abstract class, this pattern makes the class-source extend the class-destination, and makes stubs for all of the constructors found in the class-destination. These constructor stubs call the corresponding constructors in the class-destination.

You can find the Implementation link and stub pattern in the Pattern Wizard by clicking the Link by Pattern or Node by Pattern buttons in the Tool PaletteToolbox, or by using the Create by Pattern context menu for a class.

The Implementation link and stub pattern creates the following members of interfaces and abstract classes:

- Methods
- Functions
- Subroutines
- Properties
- Indexers
- Events

See Also

Together Pattern Procedures (see page 251)

Pattern Organizer (see page 1107)

Pattern Registry (↗ see page 1109)

1.7.16 Refactoring Overview

Together provides extensive support for refactoring your implementation projects.

Refactoring means rewriting existing source code with the intent of improving its design rather than changing its external behavior. The focus of refactoring is on the structure of the source code, changing the design to make the code easier to understand, maintain, and modify.

The refactoring features provided by Together affect both source code and model. As a result, your project is consisting after refactoring, even if it includes UML diagrams.

The primary resource book on refactoring is *Refactoring - Improving the Design of Existing Code by Martin Fowler (Addison - Wesley, 1999)*.

See Also

Using Refactor Operations (↗ see page 184)

Refactor Operations Reference (↗ see page 1115)

1.7.17 Quality Assurance Facilities Overview

Together provides audits and metrics as Quality Assurance features to unobtrusively help you enforce company standards and conventions, capture real metrics, and improve what you do. Although audits and metrics are similar in that they both analyze your code, they serve different purposes.

Audits and metrics are run as separate processes. Because the results of these two processes are different in nature,Together provides different features for interpreting and organizing the results. Note that some of the features and procedures described in this section apply to both audits and metrics while some are specific to one or the other.

Audits

When you run audits, you select specific rules to which your source code should conform. The results display only the violations of those rules so that you can examine each problem and decide whether to correct the source code. Together provides a wide variety of audits to choose from, ranging from design issues to naming conventions, along with descriptions of what each audit looks for and how to fix violations. You can create, save, and reuse sets of audits to run. Together ships with a predefined saved audit set `(current.adt)` and you can create your own custom sets of audits to use.

Warning: This feature is available for implementation projects only.

Metrics

Metrics evaluate object model complexity and quantify your code. It is up to you to examine the results and decide whether they are acceptable. Metrics results can highlight parts of code that need to be redesigned, or they can be used for creating reports and for comparing the overall impact of changes in a project.

Together supports a wide range of metrics. See the descriptions of available metrics in the Metrics dialog window.

You can define, save, and reuse sets of metrics.

Along with the full set of metrics, Together provides tips for using metrics and interpreting results.

Warning: This feature is available for implementation projects only.

Bar chart

Metrics results can also be viewed graphically. Two graphic views allow you to summarize metrics results: bar charts and Kiviat charts. Both charts are invoked from the context menu of the table. Use the Kiviat chart for rows and the bar chart for columns.

The bar chart displays the results of a selected metric for all packages, classes, and/or operations.

The bar color reflects conformance to the limiting values of the metric in reference:

- Green represents values that fall within the permissible range.
- Red represents values that exceed the upper limit.
- Blue represents values that are lower than the minimal permissible value.
- A thin vertical red line represents the upper limit and a thin vertical blue line represents the lower limit.

Kiviat chart

Use the Kiviat chart for rows and the bar chart for columns.

The Kiviat chart demonstrates the analysis results of the currently selected class or package for all the metrics that have predefined limiting values. The metrics results are arranged along the axes that originate from the center of the graph.

Each axis has a logarithmic scale with the logarithmic base being the axis metric upper limit so that all upper limit values are equidistant from the center. In this way, limits and values are displayed using the following notation:

- Upper limits are represented by a red circle. Any points outside the red circle violate the upper limit.
- Lower limits are represented by blue shading, showing that any points inside the blue area violate the lower limit. Note that blue shading does not show up in areas of the graph with lower limits of 1 or 0.

As the mouse cursor hovers over the chart, the Visual Studio status bar displays information about the metrics or metrics values that correspond to the tick marks.

- The actual metrics show up in the form of a star with metric values drawn as points.
- Green points represent acceptable values.
- Blue points represent values below the lower limit.
- Red points represent values exceeding the upper limit.
- Scale marks are displayed as clockwise directional ticks perpendicular to the Kiviat ray.
- Lower limit labels are displayed as counterclockwise directional blue ticks perpendicular to the Kiviat ray.

Sets of audits and metrics

Both Audits and Metrics dialog boxes display the set of all available audits and metrics. When you open a project, a default subset is active. Active audits and metrics are indicated by checkmarks. If you open the desired dialog and click Start, all of the active audits/metrics are processed.

You will not want to run every audit or metric in the default active set every time, but rather some specific subset. Together enables you to create saved sets of active audits and metrics that can be loaded and processed as you choose. To do that, use the Load Set and Save Set buttons on the toolbar of the Audits and Metrics dialog windows. You can always restore the default active set using the Set Defaults button in the Audits dialog. Refer to the Audits dialog for description of controls.

Use the default active audits set or any saved set as the basis for creating a new saved set. By default, audit sets are saved in the QA folder under the Together installation.

See Also

Running Audits (↗ see page 273)

Running Metrics (↗ see page 276)

1.7.18 Documentation Generation Facility Overview

This feature automatically generates documentation for your project. Use this feature to illustrate you programme with the documentation in the HTML format. You can update this automatically generated documentation when your project changes, or edit this documentation manually afterwards.

Documentation files

All the documentation that Together generates is written to a single directory that you specify in the `Output` folder of the **Generate HTML** dialog box. By default, the generated documentation opens in your external web browser. The browser opens with a frameset to display the generated documentation. If you choose not to open the documentation immediately, you can open it later using the `index.html` file found on the root of the documentation directory specified in the **Generate HTML** dialog box.

HTML documentation frames

The HTML documentation contains three frames:

- **Diagram frame**, when **Include diagrams** option is turned on
- **Project** and **Overview frame**, when **Include navigation tree** option is turned on
- **PackageList** and **PackageOverview frame**, when **Include navigation tree** option is turned off
- **Documentation frame**

You can click the **Project** tab in the lower left frame and expand the nodes in the project tree view. Notice that clicking a class name in the Project tab opens the documentation in the lower right pane (the **Documentation frame**). When you select a diagram in the **Project** tab, it opens in the **Diagram frame**. Elements in the **Diagram** frame are hyperlinked to the **Documentation** frame. If you select an element in the **Diagram** frame, its contents are displayed in the **Documentation** frame.

The **Documentation** frame displays the documentation of your source code and diagrams, and includes everything you would expect when generating HTML documentation. The top of the **Documentation** frame contains a navigation bar for browsing your project documentation.

The **Project** tab contains a tree representation of the project. Expand the nodes to reveal individual diagrams and elements. Clicking a class or interface opens the related documentation in the **Documentation** frame.

See Also

Generating Project Documentation (↗ see page 248)

Configuring the Documentation Generation Facility (↗ see page 247)

1.7.19 Model Import and Export Overview

You can share model information with other systems by importing and exporting model information, or by sharing project files:

Import and export features

Feature	Description
Exporting diagrams to images	You can save diagrams in several formats, including: Bitmap image (`BMP`) Enhanced windows metafile (`EMF`) Graphics interchange (`GIF`) JPEG file interchange (`JPG`) W3C portable network graphics (`PNG`) Tag image file (`TIFF`) Windows metafile (`WMF`)
Importing IBM Rational Rose (MDL) models	It is possible to convert models designed in IBM Rational Rose 2003 to the format of Together. The following file formats are supported: `.mdl, .ptl, .cat`, and `.sub`. For import, you create a new design UML 1.5 project based on the IBM Rational Rose project.
Importing from XMI Exporting to XMI	XMI (XML Metadata Interchange) enables the exchange of metadata information. Using XMI, you can exchange models across languages and applications. For example, if you have a modeling project created with a tool other than Together, you can import it to Together as an XMI file for extension or as the basis of a new project. Likewise, you can export Together projects for use in other applications. The result in each case is a single, portable .xml file. Together supports UML 1.3 Unisys XMI interchange for 8 types of UML diagrams. This feature is available for design projects that comply with the UML 1.5 specification.
Importing from other versions of Together Sharing with other versions of Together	You can reuse models created in other editions and versions of **Borland Together**. This feature is known as **interoperability**.
Export a Quality Assurance metric chart to image	Create a chart and then export it to image.

See Also

Interoperability Overview

Exporting Diagram to Image (see page 197)

Importing a Project in IBM Rational Rose (MDL) Format (see page 265)

Importing a Project to XMI Format (see page 266)

Exporting a Project to XMI Format (see page 264)

Creating a Chart (see page 275)

Supported Project Formats (see page 1116)

2 Procedures

This section provides how-to information for various areas of RAD Studio development.

Topics

Name	Description
Compiling and Building Procedures (↗ see page 104)	This section provides how-to information on building packages and localizing applications.
Debugging Procedures (↗ see page 114)	This section provides how-to information on debugging applications.
Deploying Applications (↗ see page 133)	This section provides how-to information on deploying applications. There are two ways to deploy an application — manually, or using the Deployment Manager. Currently, the Deployment manager is only for use with ASP.NET applications. Manual procedures for deploying applications are described in the appropriate Win32 area in this help system (for example, Delphi, ECO, or COM).
Editing Code Procedures (↗ see page 136)	This section provides how-to information on using the features of the **Code Editor**.
Getting Started Procedures (↗ see page 151)	This section provides how-to information on configuring the IDE, working with forms and projects, and more.
Localization Procedures (↗ see page 169)	This section provides how-to information on localizing applications by using the RAD Studio translation tools.
Managing Memory (↗ see page 175)	This section provides how-to information on using the Memory Manager, covering how to configure the Memory Manager, increase the memory address space, monitor the Memory Manager, use the memory map, share memory, and report and manage memory leaks.
Unit Test Procedures (↗ see page 180)	This section provides how-to information on using the features of DUnit and NUnit.
Together Procedures (↗ see page 183)	This section provides how-to information on using the Together features.

2.1 Compiling and Building Procedures

This section provides how-to information on building packages and localizing applications.

Topics

Name	Description
Applying the Active Build Configuration for a Project (↗ see page 104)	
Building Packages (↗ see page 105)	You can create packages in RAD Studio and include them in your projects.
Compiling C++ Design-Time Packages That Contain Delphi Source (↗ see page 106)	C++Builder supports compiling design-time packages that contain Delphi source files. However, if any of those Delphi sources make reference to IDE-supplied design-time units such as DesignIntf, DesignEditors, and ToolsAPI that exist in DesignIDE100.bpl, you must take steps to ensure that the references can be resolved by the C++Builder package.
Creating Build Events (↗ see page 107)	You can use the build events dialog to create a list of events that occur in various stages of the build process. Depending on the type of project, you can create events for the pre-build, pre-link, and post-build stages. You add events for any of these stages in exactly the same way.
Creating Named Build Configurations for C++ (↗ see page 107)	
Creating Named Build Configurations for Delphi (↗ see page 108)	
Building a Project Using an MSBuild Command (↗ see page 108)	The IDE uses Microsoft's MSBuild engine to build a project. You can build projects without knowing anything about MSBuild; the IDE handles all the details for you. However, you can also directly build the project using MSBuild command-line syntax as described here. When you build a project, the results of the build appear in the **Output** pane of the **Messages** window. If you have entered build events, the build output pane displays the commands you specified and their results. MSBuild command-line syntax has the form: `MSBuild <projectname> [/t:<target name>][/p:configuration=<configuration name>]`
Using Targets Files (↗ see page 109)	
Installing More Computer Languages (↗ see page 110)	If you have installed RAD Studio with only one or two computer languages (Delphi, C#, C++), and you later decide to add a language that was not originally installed, follow the steps below.
Linking Delphi Units Into an Application (↗ see page 111)	When compiling an application that references a Delphi-produced assembly, you can link the Delphi units for that assembly into your application. The compiler will link in the binary DCUIL files, which will eliminate the need to distribute the assembly with your application.
Previewing and Applying Refactoring Operations (↗ see page 111)	You can preview most refactoring operations in the **Refactoring** pane. Some refactorings occur immediately and allow no preview. You might want to use the preview feature when you first begin to perform refactoring operations. The preview shows you how the refactoring engine evaluates and applies refactoring operations to various types of symbols and other refactoring targets. Previewing is set as the default behavior. When you preview a refactoring operation, the engine gathers refactoring information in a background thread and fills in the information as the information is collected. If you apply a refactoring operation right away, it is performed in... more (↗ see page 111)
Renaming a Symbol (↗ see page 112)	You can rename symbols if the original declaration symbol is in your project, or if a project depended upon by your project contains the symbol and is in the same open project group. You can also rename error symbols.
Working with Named Option Sets (↗ see page 113)	

2.1.1 Applying the Active Build Configuration for a Project

To use the Configuration Manager to apply an active build configuration to a project or projects

1. Open the Configuration manager either by choosing Project ▶ Configuration Manager or by right-clicking a project group in the **Project Manager** and selecting **Configuration Manager** from the context menu.

2. On the **Build Configuration Manager**, select a configuration from the **Configuration name** list. The list includes all the available configurations you have created in the project group, including the default configurations (**Debug** and **Release**).
3. In **Available projects**, select the project or projects that you want to use the selected configuration. Selecting a configuration name causes the Available projects list to display the names of the projects that are associated with that configuration. The Available projects list also includes the **Active Configuration** for each project.
4. To designate the selected **Configuration name** as the active configuration for the projects you've selected, click **Apply**.

   Tip: To select all the available projects, click Select All

   .

   Tip: To clear all selections in the Available Projects

   list, click **Clear All**.

To use the Project Manager to activate a build configuration

1. In the **Project Manager**, either double-cliick the name of a build configuration (such as Debug or Release) or right-click the name and select **Activate** from the context menu.
2. The name of the configuration changes to boldface to indicate that it is now the current active build configuration.

See Also

MSBuild Overview (see page 4)

Build Configurations Overview (see page 5)

Compiling (see page 2)

Creating Named Build Configurations for C++ (see page 107)

Build Configuration Manager (see page 828)

2.1.2 Building Packages

You can create packages in RAD Studio and include them in your projects.

To create a new package

1. **File ▶ New ▶ Other** to display the **New Items** object gallery.
2. Depending on your type of project, select either the **Delphi Projects** node, the **Delphi for .NET Projects** node, or the **C++Builder Projects** node.
3. Double-click the **Package** icon. This creates a new empty package and makes an entry for it in the **Project Manager**, along with two folders: one marked **Contains** and one marked **Requires**.

   Note: If you want to add required files to the package, you must add compiled packages (.dcpil

   , `.dll`) to the **Required** folder. Add uncompiled code files (`.pas`, `.cpp`, `.h`) to the **Contains** folder.
4. Select the package name in the Project Manager.
5. Right-click to display the drop-down context menu and choose Add to display the **Add** dialog box.
6. Browse to locate the file or files you want to add.
7. Select one or more files, and click **Open**.
8. Click **OK**. This adds the selected files to the package.
9. Choose **Project ▶ Build <Package Name>** to build the package.

To add a package to a project

1. Choose **File ▶ New ▶ Other ▶ VCL Forms Application**.
2. Select the project name in the **Project Manager**.
3. Right-click to display the drop-down context menu.
4. Choose **Add**.
5. Browse to locate a package file.
6. Select the file and click **Open**.
7. Click **OK**. This adds the package to the project.
8. Choose **Project ▶ Build <Project Name>** to build the project.

To add a component package to the Tool Palette

1. Choose **Components ▶ Installed .NET Components**.
2. Click the **.NET VCL Components** tab.
3. Click **Add**.
4. Locate the package file you want to add to the **Tool Palette**.
5. Click **Open**. This displays the available components from the package.
6. Click **OK**. The components appear in the **Tool Palette**.

See Also

Compiling (see page 2)

2.1.3 Compiling C++ Design-Time Packages That Contain Delphi Source

C++Builder supports compiling design-time packages that contain Delphi source files. However, if any of those Delphi sources make reference to IDE-supplied design-time units such as DesignIntf, DesignEditors, and ToolsAPI that exist in DesignIDE100.bpl, you must take steps to ensure that the references can be resolved by the C++Builder package.

To compile design-time packages that contain Delphi source

1. The Delphi compiler must be able to resolve units in the DesignIDE package. To enable this, in the C++Builder package project, go to **Project ▶ Options ▶ Delphi Compiler ▶ Other Options**.
2. Add `–LUDesignIDE` to the **Additional Options** field.
3. The C++ linker must be able to resolve the reference inside the compiled design-time .obj to link. To allow this, in the C++Builder package project, select the **Project ▶ Add To Project ▶ Requires** tab.
4. In the **Package name** field, enter `designide.bpi` and click **OK**.

You can now compile and install your C++Builder package.

See Also

Design-time Packages

2.1.4 Creating Build Events

You can use the build events dialog to create a list of events that occur in various stages of the build process. Depending on the type of project, you can create events for the pre-build, pre-link, and post-build stages. You add events for any of these stages in exactly the same way.

To create a list of build events

1. Choose **Project** ▶ **Options** ▶ **Build Events**.
2. Click **Commands** in the build phase to which you want to add events (**Pre-Build**, **Pre-Link**, or **Post-Build**).
3. Enter the build commands, one command per line, and press `Return` after entering each command. Commands consist of any valid DOS command, such as: `copy $(<OutputPath>) c:\Built\$(<OutputName>)`
4. To display the **Events List** dialog box to help you enter commands, click **Edit.** In the **Commands** box on the **Events List** dialog box, enter build commands. You can edit the commands in this box.
5. To add a predefined macro on the **Events List** dialog box, scroll to the desired macro in the **Macros** list and double-click the macro name.
6. After you have finished entering commands in the **Events List** dialog box, click **OK**.
7. After you have finished entering commands for all the build phases, click **OK**.
8. The commands and their results are displayed in the **Output** pane of the **Messages** window.

See Also

Build Events Dialog Box (see page 829)

Pre-Build Event or Post-Build Event Dialog Box (see page 844)

2.1.5 Creating Named Build Configurations for C++

To create a new build configuration

1. In the Project Manager, locate the **Build configurations** node. This node represents the settings of the **Base** build configuration. The node lists all the build configurations.
2. To create a new build configuration based on an existing build configuration, right-click the name of the configuration (such as **Debug** or **Release**) and select **Add new**.
3. The new configuration appears in the Project Manager and is named Configuration1 if it is the first new configuration you create. New configurations are numbered sequentially.
4. To rename the new configuration, right-click the name and select Rename.

To change an existing build configuration

1. Choose **Project** ▶ **Options** and locate **Build configuration**, the first field on all the build-related pages, such as the **Paths and Defines** page.
2. On any of the pages that contains the **Build configuration** field, select the name of the build configuration you want to change.
3. Adjust the project options on each of the build-related pages as appropriate for your needs.
4. Click **OK** to save

Tip: To apply a build configuration to a project or project group, either right-click a configuration in the Project Manager and select Activate, or choose **Project ▶ Configuration Manager** and click **Apply**.

See Also

MSBuild Overview (see page 4)

Build Configurations Overview (C++) (see page 6)

Build Configuration Manager (see page 828)

Applying the Active Build Configuration (see page 104)

Working With Named Option Sets (C++) (see page 113)

Setting C++ Project Options (see page 163)

2.1.6 Creating Named Build Configurations for Delphi

To create a new build configuration

1. In the Project Manager, locate the **Build configurations** node. All the existing build configurations are listed here.
2. Do either of the following:

- To create a new build configuration based on the current settings in the **Project ▶ Options** dialog box, right-click the Build Configurations node and select **Add new**.
- To create a new build configuration based on an existing build configuration, right-click the name of the configuration (such as **Debug** or **Release**) and select **Add new**.

To change an existing build configuration

1. Choose **Project ▶ Options**. **Build configuration** is the first field on all the build-related pages.
2. On any of the pages that contains the **Build configuration** field, select the name of the **Build configuration** you want to change.
3. Adjust the project options on each of the build-related pages as appropriate for your needs.

Tip: To apply a build configuration to a project or project group, choose Project->Configuration Manager

and click **Apply**.

See Also

MSBuild Overview (see page 4)

Build Configurations Overview (see page 5)

Build Configuration Manager (see page 828)

Applying the Active Build Configuration (see page 104)

2.1.7 Building a Project Using an MSBuild Command

The IDE uses Microsoft's MSBuild engine to build a project. You can build projects without knowing anything about MSBuild; the IDE handles all the details for you. However, you can also directly build the project using MSBuild command-line syntax as described here. When you build a project, the results of the build appear in the **Output** pane of the **Messages** window. If you have entered build events, the build output pane displays the commands you specified and their results.

MSBuild command-line syntax has the form:

```
MSBuild <projectname> [/t:<target name>][/p:configuration=<configuration name>]
```

To build a project using the command line

1. From the **Start** menu, select **CodeGear RAD Studio**▶**RAD Studio Command Prompt**. The command prompt window automatically sets the environment for using RAD Studio tools such as MSBuild.exe.
2. Navigate to the directory that contains your project, such as C:\Documents and Settings\<myname>\My Documents\RAD Studio\Projects.
3. Type `msbuild` but do not press `Return` yet.
4. Enter your project name, such as `TelePoll.dproj` (a Delphi project) or `UserInfo.cbproj` (a C++ project). If the project is not in the current directory, you must include the full path name to the project directory.
5. To specify a target, enter the `/t:` tag followed by one of the targets specified in your project file. The three standard target names are `clean`, `make`, and `build`:

- `Clean` means to clean the project, removing generated files such as object code. `Clean` corresponds to the Project Manager context menu item Clean.
- `Make` means to compile the project. `Make` corresponds to the context menu item Compile.
- `Build` means to build the project. `Build` corresponds to the context menu item Build. The three targets are similar to the Clean, Compile, and Build commands on the context menu in the **Project Manager**. The default target is `build`.

6. To specify a configuration, enter the configuration name after `/p:configuration =`. If you do not specify a configuration, MSBuild uses the current active configuration. To specify a configuration, use the name of one of the existing build configurations in your project. This can be either a default configuration, such as **Debug**, or a configuration you have added to the project. If the configuration name has a space in it, enter the name bounded by double quotes, such as: `/p:configuration =“My config”`
7. Enter any other options and press Return to begin the build.

To display online help for MSBuild (including a full example command line), open the RAD Studio Command Prompt (see Step 1) and enter `MSBuild /help`.

For more information about MSBuild, see the Microsoft documentation at `http:\\msdn.microsoft.com`.

See Also

MSBuild Overview (see page 4)

Compiling (see page 2)

IDE Command Line Switches and Options (see page 1082)

Build Configuration Manager (see page 828)

Creating Named Build Configurations

Files Generated by RAD Studio (see page 1084)

2.1.8 Using Targets Files

To create a new .targets file in the project from the menu

1. Choose **File**▶**NewOther...**.
2. In **C++Builder Projects** click **C++Builder Files**.
3. On that page, click **MSBuild Targets File**.
4. A new .targets XML file is created and displayed in the IDE window. It contains only a **<Project>** node.

To add a .targets file to the project from the menu

1. Choose Project ▶ Add To Project.
2. In the **Add to project** dialog, select **MSBuild targets file (*.targets)** from the **Files of type:**pull down menu.
3. Navigate to the .targets file.
4. Click **Open** to add the file to the project and close the dialog. Click **Cancel** to not add the file and close the dialog.

To add a .targets file to the project from the Project Manager

1. Right-click the project in the **Project Manager**.
2. Click **Add...** in the context menu.
3. In the **Add to project** dialog, select **MSBuild targets file (*.targets)** from the **Files of type:**pull down menu.
4. Navigate to the .targets file.
5. Click **Open** to add the file to the project and close the dialog. Click **Cancel** to not add the file and close the dialog.

To enable a .targets file

1. Right-click the .targets file in the **Project Manager**.
2. Click **Enable** in the context menu.

To ensure a .targets file is conformant and error free

1. Right-click the .targets file in the **Project Manager**.
2. Click **Validate** in the context menu.

To remove a .targets file from the project

1. Right-click the .targets file in the **Project Manager**.
2. Click **Remove From Project** in the context menu.

See Also

Targets files (see page 8)

2.1.9 Installing More Computer Languages

If you have installed RAD Studio with only one or two computer languages (Delphi, C#, C++), and you later decide to add a language that was not originally installed, follow the steps below.

To add more computer languages to your IDE:

1. Choose Start ▶ Settings ▶ Control Panel ▶ Add or Remove Programs.
2. Select RAD Studio
3. Click the Change button.
4. When the Installation Wizard comes up, it will ask you if you want to Modify, Repair, or Remove the program. Select the Modify radio button.
5. Follow the rest of the steps in the Installation Wizard to choose the languages that you want to add.
6. Click the Finish button.

2.1.10 Linking Delphi Units Into an Application

When compiling an application that references a Delphi-produced assembly, you can link the Delphi units for that assembly into your application. The compiler will link in the binary DCUIL files, which will eliminate the need to distribute the assembly with your application.

To link in a Delphi unit

1. With your application open in the IDE, choose **Project**▶**Add Reference**.
2. In the **Add Reference** dialog box, select a Delphi-produced assembly DLL from the list of .NET assemblies and click the **Add Reference** button. If the assembly you want to link to is not in the list, use the **Browse** button to find and select it.
3. Click **OK**. The assembly is listed in the References node of the **Project Manager**.
4. In the **Project Manager**, right-click the assembly and choose Link in Delphi Units. The menu command is disabled if the reference is not a Delphi-produced assembly. In the **Object Inspector**, the corresponding Link Units property is set to **True**.
5. Choose **Project**▶**Compile** to compile the application.

2.1.11 Previewing and Applying Refactoring Operations

You can preview most refactoring operations in the **Refactoring** pane. Some refactorings occur immediately and allow no preview. You might want to use the preview feature when you first begin to perform refactoring operations. The preview shows you how the refactoring engine evaluates and applies refactoring operations to various types of symbols and other refactoring targets. Previewing is set as the default behavior. When you preview a refactoring operation, the engine gathers refactoring information in a background thread and fills in the information as the information is collected.

If you apply a refactoring operation right away, it is performed in a background thread also, but a modal dialog blocks the UI activity. If the refactoring engine encounters an error during the information gathering phase of the operation, it will not apply the refactoring operation. The engine only applies the refactoring operation if it finds no errors during the information gathering phase.

To preview a refactoring operation

1. Open a project.
2. Locate a symbol name in the **Code Editor**.
3. Select the symbol name.
4. Right-click to display the context menu.
5. Select **Refactoring**▶**Rename 'symbol type'** where *symbol type* is one of the valid types, such as method, variable, or field. This displays the **Rename Symbol** dialog.
6. Type a new name in the **New name** text box.
7. Select the **View references before refactoring** check box.
8. Click **OK**. This displays a hierarchical list of the potentially refactored items, in chronological order as they were found. You can jump to each item in the Code Editor.

   Note: If you want to remove an item from the refactoring operation, select the item and click the Delete Refactoring

   icon in the toolbar.

2

To jump to a refactoring target from the Message Pane

1. Expand any of the nodes that appear in the **Message Pane**.
2. Click on the target refactoring operation that you would like to view in the **Code Editor**.
3. Make any changes you would like in the **Code Editor**.

   Warning: If you change an item in the Code Editor

   , the refactoring operation is prevented. You need to reapply the refactoring after making changes to any files during the process, while the **Message Pane** contains refactoring targets.

To apply refactorings

1. Open a project.
2. Locate a symbol name in the **Code Editor**.
3. Select the symbol name.
4. Right-click to display the context menu.
5. Select **Refactoring**▶**Rename 'symbol type'** where *symbol type* is one of the valid types, such as method, variable, or field. This displays the **Rename Symbol** dialog.
6. Type a new name in the **New name** text box.
7. Click **OK**. As long as the **View references before refactoring** check box is not selected, the refactoring occurs immediately.

   Warning: If the refactoring engine encounters errors, the refactoring is not applied. The errors are displayed in the Message Pane

   .

2

See Also

Refactoring Overview (↗ see page 57)

2.1.12 Renaming a Symbol

You can rename symbols if the original declaration symbol is in your project, or if a project depended upon by your project contains the symbol and is in the same open project group. You can also rename error symbols.

To rename a symbol

1. Select the symbol name in the **Code Editor**.
2. Right-click to display the drop-down context menu.
3. Select **Refactoring**▶**Rename 'symbol type' ' symbol name'** where *symbol type* is either method, variable, or field, and *symbol name* is the actual name of the selected symbol. This displays the **Rename** dialog box.
4. Enter the new name in the **New Name** text box.
5. If you want to preview the changes to your project files, select the **View References Before Refactoring** check box.

   Note: The menu commands are context-sensitive. If you select a method, the command will read **Rename Method***method name* where *method name* is the actual name of the method you have selected. This context-sensitivity holds true for all other object types, as well.

See Also

Refactoring Overview (↗ see page 57)

2.1.13 Working with Named Option Sets

To create a new named option set from Project Options

1. Choose Project ▶ Options.
2. **Build configuration** is the first field on the page. Select from the drop-down menu the configuration on which you want to base the named option set.
3. Adjust the options on any one or more of the pages associated with the build configuration.
4. Click the **Save As...** button.
5. In the **Save Options Set** dialog box, enter a name for the named option set you are creating.
6. Click **OK** to save the option set to a file.

To create a new named option set from Project Manager

1. In the **Project Manager**, right-click the configuration from which you want to save an option set.
2. On the context menu, click **Save As...**.
3. In the **Save Options Set** dialog box, enter a name for the named option set you are creating.
4. Click **OK** to save the option set to a file.

To load a named option set from Project Options

1. Choose Project ▶ Options.
2. Select the name of the **Build configuration** from the pull down menu to which you want to apply the option set.
3. Click the **Load...** button to display the **Apply Option Set** dialog.
4. Navigate to the desired file for the named option set.
5. Choose how you load the values from the named option set: **Overwrite**, **Replace**, or **Preserve**.
6. Click **OK** to apply the option set and close the dialog. Click **Cancel** to not apply it and close the dialog.

2

To load a named option set from Project Manager

1. In the **Project Manager**, right-click the configuration to which you want to apply an option set.
2. On the context menu, click **Apply Option Set...** to display the **Apply Option Set** dialog.
3. Navigate to the desired file for the named option set.
4. Choose how you load the values from the named option set: **Overwrite**, **Replace**, or **Preserve**.
5. Click **OK** to apply the option set and close the dialog. Click **Cancel** to not apply it and close the dialog.

See Also

Apply Option Set dialog (↗ see page 828)

MSBuild Overview (↗ see page 4)

C++ Build Configurations Overview (↗ see page 6)

Build Configuration Manager (↗ see page 828)

Setting C++ Project Options (↗ see page 163)

Creating Named Build Configurations (C++) (↗ see page 107)

Applying the Active Build Configuration (↗ see page 104)

2.2 Debugging Procedures

This section provides how-to information on debugging applications.

Topics

Name	Description
Adding a Watch (↗ see page 116)	Add a watch to track the values of program variables or expressions as you step over or trace into code. Each time program execution pauses, the debugger evaluates all the items listed on the Active tab (or ActiveWatchGroup) in the **Watch List** window and updates their displayed values. You can organize watches into groups. When you add a watch group, a new tab is added to the **Watch List** window and all watches associated with that group are shown on that tab. When a group tab is displayed, only the watches in that group are evaluated during debugging. By grouping... more (↗ see page 116)
Using the CPU View (↗ see page 116)	The CPU view displays the assembly language code for your program.
Displaying Expanded Watch Information (↗ see page 117)	When you debug an application, you can inspect the values of members within a watched object whose type is a complex data object (such as a class, record, or array). These values display in the **Watch List** window when you expand a watched object. Additionally, you can expand the elements within an object, displaying its sub-elements and their values. You can expand all levels in the object. Members are grouped by ancestor.
Attaching to a Running Process (↗ see page 117)	You can attach to a process that is running on your computer or on a remote computer. This is useful for debugging a program that was not created with RAD Studio.
Setting and Modifying Breakpoints (↗ see page 118)	Breakpoints pause program execution at a certain location or when a particular condition occurs. You can set source breakpoints and module load breakpoints in the **Code Editor** before and during a debugging session. You can set data breakpoints and address breakpoints only when the application is running in debug mode (F9), **Procedures described in this Topic:** • To set a source breakpoint • To set an address breakpoint • To set a data breakpoint • To set a module load breakpoint • To create a breakpoint group • To enable or disable a breakpoint or a breakpoint group • To create a conditional breakpoint • To associate actions... more (↗ see page 118)
Debugging VCL for .NET Source Code (↗ see page 121)	To debug VCL for .NET source code, you must set certain project options that are not needed when debugging other types of applications. The options are off by default and must be specifically set.
Using Tooltips During Debugging (↗ see page 122)	When you debug an application, you can display the values of members within a watched object whose type is a complex data object (such as a class, record, or array). These values display in the code editor window when you expand a watched object. Additionally, you can expand the elements within an object, displaying its sub-elements and their values. You can expand all levels in the object. Members are grouped by ancestor.
Inspecting and Changing the Value of Data Elements (↗ see page 122)	The **Debug Inspector** lets you inspect data elements by automatically formatting the type of data it is displaying. The **Debug Inspector** is especially useful for examining compound data objects, such as arrays and linked lists. Because you can inspect individual items displayed in the **Debug Inspector**, you can perform a *walkthrough* of compound data objects by opening a **Debug Inspector** on a component of the compound object. **Note:** The Debug Inspector is only available when the process is stopped in the debugger.

Modifying Variable Expressions (↗ see page 124)	After you have evaluated a variable or data structure item, you can modify its value. When you modify a value through the debugger, the modification is effective for the program run only. Changes you make through the **Evaluate/Modify** dialog box do not affect your source code or the compiled program. To make your change permanent, you must modify your source code in the **Code Editor**, then recompile your program.
Preparing a Project for Debugging (↗ see page 124)	While most debugging options are set on by default, you can use the following procedures to review and change those options. There are both general IDE options and project specific options. The project specific options vary based on the active project type, for example, Delphi, Delphi .NET, or C#.
Remote Debugging: Metaprocedure (↗ see page 125)	Remote debugging lets you debug a RAD Studio application running on a remote computer. Once the remote debug server is running on the remote computer, you can use RAD Studio to connect to that computer and begin debugging.
Installing, Starting, and Stopping the Remote Debug Server (↗ see page 125)	Remote debugging lets you debug a RAD Studio application running on a remote computer. Once the remote debug server is running on the remote computer, you can use RAD Studio to connect to that computer and begin debugging. **Prerequisites and security considerations for remote debugging** • The local and remote computers must be connected through TCP/IP. • All of the files required for debugging the application must be available on the remote computer before you begin debugging. This includes executables, DLLs, assemblies, data files, and PDB (debug) files. • In addition to the port that the remote debug server listens on, a connection... more (↗ see page 125)
Installing a Debugger on a Remote Machine (↗ see page 126)	To debug a project on a machine that does not have RAD Studio installed, you must install the remote debugger executable files. You can install these files either directly from the installation disk or by copying them from a machine that has RAD Studio installed.
Establishing a Connection for Remote Debugging (↗ see page 127)	You must establish a TCP/IP connection between the local and remote machines in preparation for remote debugging. This connection uses multiple ports that are chosen dynamically by Windows. The remote debug server listens on one port, and a separate port is opened for each application that is being debugged. A firewall that only allows connections to the listening port will prevent the remote debugger from working. **Note:** If the remote machine uses the firewall included with Windows XP service pack 2, you will receive a message asking whether CodeGear remote debugging service should be allowed. You must indicate that this... more (↗ see page 127)
Preparing Files for Remote Debugging (↗ see page 128)	Executable files and symbol files must be copied to the remote machine after they are compiled. You must set the correct options on your local machine in order to generate these files.
Setting the Search Order for Debug Symbol Tables (↗ see page 129)	Symbol tables are used internally during debugging. By default, RAD Studio locates and uses all symbol tables available. However, you can control the order in which these symbol tables are searched. You can also limit the search to specific symbol tables, which can speed up the debugging process. The extensions for symbol table files vary by personality. • Delphi Win32, does not use external symbol files because the compiler holds the symbols tables in memory. However, if you are debugging a remote application, you must generate symbol files with the .RSM extension. • Delphi.NET, VB.NET and C# symbol files use the .PDB... more (↗ see page 129)
Resolving Internal Errors (↗ see page 130)	The error message, `Internal Error: X1234` indicates that the compiler has encountered a condition, other than a syntax error, that it cannot successfully process. **Tip:** Internal error numbers indicate the file and line number in the compiler where the error occurred. This information may help Technical Support services track down the problem. Be sure to record this information and include it with your internal error description.

2.2.1 Adding a Watch

Add a watch to track the values of program variables or expressions as you step over or trace into code. Each time program execution pauses, the debugger evaluates all the items listed on the Active tab (or ActiveWatchGroup) in the **Watch List** window and updates their displayed values.

You can organize watches into groups. When you add a watch group, a new tab is added to the **Watch List** window and all watches associated with that group are shown on that tab. When a group tab is displayed, only the watches in that group are evaluated during debugging. By grouping watches, you can also prevent out-of-scope expressions from slowing down stepping.

To add a watch

1. Choose **Run**▶**Add Watch** to display the **Watch Properties** dialog box.
2. In the **Expression** field, enter the expression you want to watch. An expression consists of constants, variables, and values contained in data structures, combined with language operators. Almost anything you can use as the right side of an assignment operator can be used as a debugging expression, except for variables not accessible from the current execution point.
3. Optionally, enter a name in the **Group Name** field to create the watch in a new group, or select a group name from the list of previously defined groups.
4. Specify other options as needed (click **Help** on the **Watch Properties** dialog for a description of the options). For example, you can request the debugger to evaluate the watch, even if doing so causes function calls, by selecting the **Allow Function Calls** option.
5. Click **OK**.

The watch is added to the **Watch List** window.

See Also

Watch Properties (see page 948)

Watch List Window (see page 1031)

2.2.2 Using the CPU View

The CPU view displays the assembly language code for your program.

To use the CPU view

1. Run your program.
2. Choose **Run**▶**Program Pause** from the menu. The CPU view displays. Note that up to five separate panes might display. Click the *CPU Window* link at the end of this topic for information about these panes.

See Also

Debugging Applications (see page 10)

CPU Window (see page 1022)

2.2.3 Displaying Expanded Watch Information

When you debug an application, you can inspect the values of members within a watched object whose type is a complex data object (such as a class, record, or array). These values display in the **Watch List** window when you expand a watched object. Additionally, you can expand the elements within an object, displaying its sub-elements and their values. You can expand all levels in the object. Members are grouped by ancestor.

To display expanded watch information in the Watch List window

1. Set a breakpoint on a valid source line within your project. A breakpoint icon displays in the gutter next to the selected line.
2. Choose **Run**▶**Add Watch** to add a watch for an object in your application. The watch displays in the **Watch List** window.
3. Choose **Run**▶**Run** to begin running the program. If needed, use the feature of the program that will cause it to run to the breakpoint you set. The IDE automatically switches to the Debug layout and the program stops at the breakpoint.
4. Click the **+** next to the name of the object that you added to the watch list. The names and values of elements of the watched object display in the **Watch List** window.

See Also

Debugging Applications (see page 10)

Adding a Watch (see page 116)

Setting and Modifying Source Breakpoints (see page 118)

Inspecting and Changing the Value of Data Elements (see page 122)

2.2.4 Attaching to a Running Process

You can attach to a process that is running on your computer or on a remote computer. This is useful for debugging a program that was not created with RAD Studio.

To attach to a running process

1. Choose **Run**▶**Attach to Process** to display the **Attach to Process** dialog box.
2. Select either **CodeGear .NET Debugger** or **CodeGear Win32 Debugger** from the **Debugger** drop-down list, depending on whether you want to attach to a .NET or Win32 process. The list of **Running Processes** is refreshed to display the appropriate processes. For Win32 processes, you can also check **Show System Processes** to include system processes in the list.
3. If the process is running on a remote computer, enter the name the computer in the **Remote Machine** field

   Note: The remote debug server must be running on the remote computer.

4. Select a process from the list of **Running Processes**.
5. If you do not want the process to pause after you have attached to it, uncheck **Pause After Attach**.
6. Click **Attach**.

See Also

Attach to Process (see page 941)

2.2.5 Setting and Modifying Breakpoints

Breakpoints pause program execution at a certain location or when a particular condition occurs. You can set source breakpoints and module load breakpoints in the **Code Editor** before and during a debugging session. You can set data breakpoints and address breakpoints only when the application is running in debug mode (F9),

Procedures described in this Topic:

- To set a source breakpoint
- To set an address breakpoint
- To set a data breakpoint
- To set a module load breakpoint
- To create a breakpoint group
- To enable or disable a breakpoint or a breakpoint group
- To create a conditional breakpoint
- To associate actions with a breakpoint
- To change the colors associated with breakpoints

During a debugging session, any line of code that is eligible for a breakpoint is marked with a blue dot • in the left gutter of the **Code Editor**.

You can also set breakpoints on frames displayed in the **Call Stack** window. The breakpoint icons in the **Call Stack** window are similar to those in the **Code Editor**, except that the blue dot • indicates only that debug information is available for the frame, not whether a breakpoint can be set on that frame.

Breakpoints are displayed in the **Breakpoint List** window, available by selecting **View ▶ Debug windows ▶ Breakpoints**.

The following icons are used to represent breakpoints in the **Code Editor** gutter.

Icon	Description
🔴	The breakpoint is valid and enabled. The debugger is inactive.
🔴✓	The breakpoint is valid and enabled. The debugger is active.
🔴✗	The breakpoint is invalid and enabled. The breakpoint is set at an invalid location, such as a comment, a blank line, or invalid declaration.
○	The breakpoint is valid and disabled. The debugger is inactive.
○✓	The breakpoint is valid and disabled. The debugger is active.
⊗	The breakpoint is invalid and disabled. The breakpoint is set at an invalid location.

To set a source breakpoint

1. To prefill the line number in the dialog box, click the line of source in the **Code Editor** at the point where you want to stop execution.
2. Choose **Run ▶ Add Breakpoint ▶ Source Breakpoint** to display the **Add Source Breakpoint** dialog box.

   Tip: To change the Code Editor

gutter, choose **Tools ▶ Options ▶ Editor Options ▶ Display** and adjust the **Gutter width** option.

3. In the **Add Source Breakpoint** dialog box, the file name is prefilled with the name of the file, and **Pass count** is set to 0 (meaning that the breakpoint fires on the first pass). In the **Line number** field, enter the line in the **Code Editor** where you want to set the breakpoint.
4. To apply a condition to the address breakpoint, enter a conditional expression in the **Condition** field. The conditional expression is evaluated each time the breakpoint is encountered, and program execution stops when the expression evaluates to True.
5. To associate the source breakpoint with a breakpoint group, enter the name of a group or select from the **Group** dropdown list.
6. To set any of the **Advanced** options, see the help topic (see page 938) for the **Add Address Breakpoint or Add Data Breakpoint** dialog box.

   Note: To quickly set a breakpoint on a line of source code, click the left gutter of the Code Editor

   next to the line of code where you want to pause execution.

To set an address breakpoint

1. Choose **Run ▶ Add Breakpoint ▶ Address Breakpoint** to display the **Add Address Breakpoint** dialog box.
2. In the **Address** field, enter the address in memory (such as $00011111) at which you want to set the breakpoint.
3. To apply a condition to the address breakpoint, enter a conditional expression in the **Condition** field. The conditional expression is evaluated each time the breakpoint is encountered, and program execution stops when the expression evaluates to true.
4. To specify that the address breakpoint will only fire after a number of passes, enter the number in the **Pass count** field.
5. To associate the address breakpoint with an existing breakpoint group, enter the group name in the **Group** field, or select the name of an existing group from the dropdown list.
6. To set any of the **Advanced** options, see the help topic (see page 938) for the **Add Address Breakpoint or Add Data Breakpoint** dialog box.

   Note: You can also set an address breakpoint in the CPU view

   or the **Disassembly view** by clicking in the gutter.

To set a data breakpoint

1. The application must be running in debug mode (for example, use F9, F8, F7, or F4).
2. Choose **Run ▶ Add Breakpoint ▶ Data Breakpoint** to display the **Add Data Breakpoint** dialog box.
3. In the **Address** field, enter the address of the data you want to function as the data breakpoint.
4. In the **Length** field, specify the length of the data operand that is to constitute the breakpoint. Note that a warning is displayed for the following issues:

- The length of the data breakpoint should not cross an even-byte boundary. (A data breakpoint with a 1-byte length has no alignment problems, but 2-byte and 4-byte data breakpoints might cover more or fewer addresses than you intend.)
- The data breakpoint should not be set on a stack location. (The breakpoint might be hit so often that the program cannot run properly.)

5. To apply a condition to the address breakpoint, enter a conditional expression in the **Condition** field. The conditional expression is evaluated each time the breakpoint is encountered, and program execution stops when the expression evaluates to true.
6. To specify that the address breakpoint only fires after a number of passes, enter the number in the **Pass count** field.
7. To associate the data breakpoint with an existing breakpoint group, enter the group name in the **Group** field, or select the name of an existing group from the dropdown list.
8. To set any of the **Advanced** options, see the help topic (see page 938) for the **Add Address Breakpoint or Add Data Breakpoint** dialog box.

To set a module load breakpoint

1. Choose **Run**▶**Add Breakpoint**▶**Module Load Breakpoint** to display the **Add Module Load Breakpoint** dialog box.
2. In the **Module name** field, enter the name of the DLL, package, or other module type that you want to monitor, or select a name from the drop down list.

   Note: You can also use the Modules view

   to set a module load breakpoint.

When the module you specify is loaded during program execution, the breakpoint is hit and program execution pauses.

To modify a breakpoint

1. Open the **Breakpoints List** by selecting **View**▶**Debug Windows**▶**Breakpoints**. Right-click the icon for the breakpoint you want to modify. For a source breakpoint, you can right-click the breakpoint icon in the **Code Editor** gutter, and choose Breakpoint Properties.
2. Set the options in the **Breakpoint Properties** dialog box to modify the breakpoint. For example, you can set a condition, create a breakpoint group, or specify an action that is to occur when execution reaches the breakpoint.
3. Click **Help** for more information about the options on the dialog box.
4. Click **OK**.

To create a breakpoint group

1. Open the **Breakpoints List** by choosing **View**▶**Debug Windows**▶**Breakpoints**.
2. Right-click the breakpoint and choose Breakpoint Properties.
3. To create a breakpoint group, enter a group name in the **Group** field. To add the breakpoint to an existing group, select a name from the dropdown list box.
4. Click **OK**.

To enable or disable a breakpoint or a breakpoint group

1. Right-click the breakpoint icon in the **Code Editor** or in the **Breakpoint List** window and choose Enabled to toggle between enabled and disabled. In the **Breakpoint List**, you can click the checkbox at the left of the icon.
2. To enable or disable all breakpoints, right-click a blank area (not on a breakpoint) in the **Breakpoint List** window and choose Enable All or Disable All.
3. To enable or disable a breakpoint group, right-click a blank area (not on a breakpoint) in the **Breakpoint List** window and choose Enable Group or Disable Group.

   Tip: Press the Ctrl

   key while clicking a breakpoint in the **Code Editor** gutter to toggle between enabled and disabled. Disabling a breakpoint or breakpoint group prevents it from pausing execution, but retains the breakpoint settings, so that you can enable it later.

To create a conditional breakpoint

1. Choose **Run**▶**Add Breakpoint** and select the type of breakpoint you want from the submenu.
2. Complete the fields in the dialog box as described in the procedure given earlier for that breakpoint type.
3. In the **Condition** field, enter a conditional expression to be evaluated each time this breakpoint is encountered during program execution. The breakpoint pauses execution when the expression evaluates to True.
4. Complete other fields as appropriate.
5. Click **OK**.

Conditional breakpoints are useful when you want to see how your program behaves when a variable falls into a certain range or what happens when a particular flag is set.

If the conditional expression evaluates to true (or not zero), the debugger pauses the program at the breakpoint location. If the expression evaluates to false (or zero), the debugger does not stop at the breakpoint location.

To associate actions with a breakpoint

1. On the **Breakpoint List**, right-click the breakpoint and choose Breakpoint Properties.
2. Click **Advanced** to display additional options.
3. Check the actions that you want to occur when the breakpoint is encountered. For example, you can specify an expression to be evaluated and write the result of the evaluation to the **Event Log**.
4. Click **OK**.

To change the color of the text at the execution point or the color of breakpoints

1. Choose **Tools**▶**Options**▶**Editor Options**▶**Color**.
2. In the code sample window, select the appropriate language tab. For example, to change the breakpoint color for Delphi code, select the Delphi tab.
3. Scroll the code sample window to display the execution and breakpoint icons in the left gutter of the window.
4. Click anywhere on the execution point or breakpoint line that you want to change.
5. Use the **Foreground Color** and **Background Color** dropdown lists to change the colors associated with the selected execution point or breakpoint.
6. Click **OK**.

   Note: You can also set breakpoints in the Breakpoint List

   , the **CPU** window (and the **Disassembly** view), the **Call Stack** view, and the **Modules** window.

See Also

Add Address Breakpoint or Add Data Breakpoint dialog box (↗ see page 938)

Add Module Load Breakpoint dialog box (↗ see page 1019)

Add Source Breakpoint dialog box (↗ see page 940)

Breakpoint List window (↗ see page 1019)

CPU window (↗ see page 1022)

Modules window (↗ see page 1028)

Call Stack window (↗ see page 1021)

2.2.6 Debugging VCL for .NET Source Code

To debug VCL for .NET source code, you must set certain project options that are not needed when debugging other types of applications. The options are off by default and must be specifically set.

To enable options for debugging VCL for .NET source code

1. Open a VCL for .NET project.
2. Choose **Project**▶**Options**▶**Compiler**.
3. Check the **Use debug DCUILs** check box.
4. Click **OK**.
5. Select any Borland-produced assembly under **References** in the **Project Manager**.

6. Right-click the assembly and choose Link in Delphi Units. This sets the **Link Units** property to **True** in the **Object Inspector**.
7. Repeat the previous two steps for each CodeGear assembly that you want to debug.

You are now able to debug VCL for .NET source code.

> **Tip:** You can use this procedure to debug VCL for .NET assemblies produced by a third party if the debug DCUILs for those assemblies are available.

See Also

Linking Delphi Units Into an Application (see page 111)

.NET Assemblies (see page 900)

Compiler (see page 831)

Project Manager (see page 1038)

2.2.7 Using Tooltips During Debugging

When you debug an application, you can display the values of members within a watched object whose type is a complex data object (such as a class, record, or array). These values display in the code editor window when you expand a watched object. Additionally, you can expand the elements within an object, displaying its sub-elements and their values. You can expand all levels in the object. Members are grouped by ancestor.

To expand tooltips during debugging

1. Create a new VCL for Win32 application or open an existing application.
2. Choose **Project ▶ Options ▶ Compiler** and verify that the **Use debug DCUs** option is selected.
3. Choose **Tools ▶ Options ▶ Editor Options ▶ Code Insight** and verify that the **Tooltip expression evaluation** option is selected.
4. Choose **Run ▶ Step Over**.

   Tip: Alternatively, press F8

   . This opens the **Code** page of the main source file for the project.
5. Choose **Run ▶ Step Over** again. This initializes the project.
6. Move the cursor over the **Application** keyword. This displays the tooltip in a single block.
7. Click the **+** next to the **Application** keyword within the tooltip. The tooltip expands to a scrollable box that contains each child property and its value. The **+** appears next to each property that has one or more child properties. You can expand any member to display properties and values hierarchically within the tooltip.

See Also

Debugging Applications (see page 10)

Compiler (see page 831)

Code Insight (see page 988)

2.2.8 Inspecting and Changing the Value of Data Elements

The **Debug Inspector** lets you inspect data elements by automatically formatting the type of data it is displaying. The **Debug**

Inspector is especially useful for examining compound data objects, such as arrays and linked lists. Because you can inspect individual items displayed in the **Debug Inspector**, you can perform a *walkthrough* of compound data objects by opening a **Debug Inspector** on a component of the compound object.

Note: The Debug Inspector

is only available when the process is stopped in the debugger.

To inspect a data element directly from the Code Editor

1. In the **Code Editor**, place the insertion point on the data element that you want to inspect.
2. Right-click and choose **Debug ▶ Inspect** to display the **Debug Inspector**.

To inspect a data element from the menu

1. Choose **Run ▶ Inspect** to display the **Inspect** dialog box.
2. In the **Inspect** dialog box, type the expression you want to inspect.
3. Click **OK**. The **Debug Inspector** is displayed.

Unlike watch expressions, the scope of a data element in the **Debug Inspector** is fixed at the time you evaluate it. If you use the Inspect command from the **Code Editor**, the debugger uses the location of the insertion point to determine the scope of the expression you are inspecting. This makes it possible to inspect data elements that are not within the current scope of the execution point.

If you use **Run ▶ Inspect**, the data element is evaluated within the scope of the execution point.

If the execution point is in the scope of the expression you are inspecting, the value appears in the **Debug Inspector**. If the execution point is outside the scope of the expression, the value is undefined and the **Debug Inspector** becomes blank.

To view members of the object you are inspecting

1. Click the **Data** tab to view strings, boolean values, and other values for such things as variable name, expression, and owner.

   Tip: If you want to see the hexadecimal representation of a string, sub-inspect the string value in the Debug Inspector

   .
2. Click the **Methods** tab to view all of the methods that are members of the object's class.

   Tip: If you want to see the return type for any method, select the method and look at the status bar of the Debug Inspector

   , where the syntax line for the method, including the return type is displayed.
3. Click the **Properties** tab to view all of the properties for the active object.
4. Click any property name to see its type displayed in the status bar of the **Debug Inspector**.
5. Click the question mark (?) icon to see the actual value for that property at this point of the execution of the application.

To change the value of a data element

1. In the **Debug Inspector**, select a data element that has an **ellipsis (…)** next to it. The ellipsis indicates that the data element can be modified.
2. Click the **ellipsis (…)**, or right-click the element and choose Change.
3. Type a new value, then click **OK**.

To inspect local variable values

1. While running in Debug mode, double-click any variable that appears in the **Local Variables** window. This displays the **Debug Inspector** for that local variable.
2. Inspect the variable's value. Change the value by clicking the button with an **ellipsis (…)** on it.

2.2.9 Modifying Variable Expressions

After you have evaluated a variable or data structure item, you can modify its value. When you modify a value through the debugger, the modification is effective for the program run only. Changes you make through the **Evaluate/Modify** dialog box do not affect your source code or the compiled program. To make your change permanent, you must modify your source code in the **Code Editor**, then recompile your program.

To change the value of an expression

1. Choose **Run ▶ Evaluate/Modify**.
2. Specify the expression in the **Expression** edit box. To modify a component property, specify the property name, for example, `this.button1.Height` or `Self.button1.Height`.
3. Enter a value in the **New Value** edit box. The expression must evaluate to a result that is assignment-compatible with the variable you want to assign it to. Typically, if the assignment would cause a compile or runtime error, it is not a legal modification value.
4. Choose **Modify**. The new value is displayed in the **Result** box. You cannot undo a change to a variable after you choose **Modify**. To restore a value, however, you can enter the previous value in the **Expression** box and modify the expression again.

   Note: You can change individual variables or elements of arrays and data structures, but you cannot change the contents of an entire array or data structure with a single expression.

   Warning: Modifying values (especially pointer values and array indexes), can have undesirable effects because you can overwrite other variables and data structures. Use caution whenever you modify program values from the debugger.

2.2.10 Preparing a Project for Debugging

While most debugging options are set on by default, you can use the following procedures to review and change those options. There are both general IDE options and project specific options. The project specific options vary based on the active project type, for example, Delphi, Delphi .NET, or C#.

To activate the integrated debugger

1. Choose **Tools ▶ Options ▶ Debugger Options**.
2. Select the **Integrated Debugging** option.
3. Click **OK**.
4. Optionally review the settings on the other debugging pages.

To set debug options

1. Choose **Project ▶ Options**.
2. Review the debugging options on the various pages of the **Project Options** dialog box. In particular, review the following pages: **Compiler**, **Linker**, **Directories/Conditionals**, **Version Info**, and **Debugger**. Note that not all pages are available for all project types. For example, the **Version Info** page is only displayed for Delphi Win32 projects.
3. Click **OK**.

See Also

Debugger Options (see page 836)

2.2.11 Remote Debugging: Metaprocedure

Remote debugging lets you debug a RAD Studio application running on a remote computer. Once the remote debug server is running on the remote computer, you can use RAD Studio to connect to that computer and begin debugging.

Use the following set of procedures to debug an application running on a remote machine

1. To enable debugging on a machine without the full IDE installation, see Installing a Debugger on a Remote Machine (see page 126)
2. To connect the local machine to the remote machine, see , Establishing a Connection for Remote Debugging (see page 127)
3. To generate program files to be copied to the remote machine, see , Preparing Files for Remote Debugging (see page 128)

See Also

Remote Debugging Overview

2.2.12 Installing, Starting, and Stopping the Remote Debug Server

Remote debugging lets you debug a RAD Studio application running on a remote computer. Once the remote debug server is running on the remote computer, you can use RAD Studio to connect to that computer and begin debugging.

Prerequisites and security considerations for remote debugging

- The local and remote computers must be connected through TCP/IP.
- All of the files required for debugging the application must be available on the remote computer before you begin debugging. This includes executables, DLLs, assemblies, data files, and PDB (debug) files.
- In addition to the port that the remote debug server listens on, a connection is opened for each application that is being debugged. Additional port numbers are chosen dynamically by Windows; a firewall that only allows connections to the listening port will prevent the remote debugger from working.

  Warning: The connection between RAD Studio and the remote debug server is a simple TCP/IP socket, with neither encryption nor authentication support. Therefore, the remote debug server should not be run on a computer that can be accessed over the network by untrusted clients.

To install and start the remote debug server

1. If RAD Studio is installed on the remote computer, skip to step 4. In this case, the remote debug server (`rmtdbg105.exe`) is already available, by default, at `C:\Program Files\CodeGear\RAD Studio\5.0\Bin`.
2. Copy `rmtdbg105.exe` from the RAD Studio`\bin` directory on your local computer to the directory of your choice on the remote computer. If you are debugging a managed application, also copy `dbkpro105.dll`
3. If you are debugging a managed application, register `dbkpro105.dll` on the remote computer by running the `regsvr32.exe` registration utility. For example, on Windows XP, enter `C:\Windows\System32\regsvr32.exe dbkdebugproide100.bpl` at the command prompt.
4. On the remote computer, run `rmtdbg105.exe` using the following syntax: `rmtdbg105.exe [-listen`

`[hostname:]port]` where:

1. `hostname` is an optional host name or TCP/IP address for binding to a particular host, for example, `somehost` or `127.0.0.1`. If you specify `hostname`, you must also specify `:port`.
2. `port` is an optional (required if `hostname` is specified) port number or standard protocol name, for example, `8000` or `ftp`. If omitted, 64447 is used as the port number. Examples:
3. `rmtdbg105.exe`
4. `rmtdbg105.exe -listen 8000`
5. `rmtdbg105.exe -listen somehost:8000`
6. `rmtdbg105.exe -listen 127.0.0.1:8000`

After the remote debug server is started, its icon appears in the Windows taskbar.

To shut down the remote debug server

1. On the remote computer, in the Windows taskbar, right-click the **CodeGear Remote Debugger Listener** icon.
2. In the shortcut menu, choose Exit.

Shutting down the remote debug server does not affect active debugging sessions.

See Also

Overview of Remote Debugging (see page 12)

Establishing a Connection for Remote Debugging (see page 127)

Preparing Files for Remote Debugging (see page 128)

2

2.2.13 Installing a Debugger on a Remote Machine

To debug a project on a machine that does not have RAD Studio installed, you must install the remote debugger executable files. You can install these files either directly from the installation disk or by copying them from a machine that has RAD Studio installed.

To install the remote debugger

1. Use the installation disk if it is available.
2. Use files from the machine that has the IDE installed if the installation disk is not available.

To install the remote debugger from the installation disk

1. Insert the installation disk into the remote machine.
2. Choose Install Remote Debugger.
3. Follow the instructions provided by the wizard.

To install the remote debugger if the installation disk is not available

1. Create a directory on the remote machine for the installation files.
2. Locate the following files on the local machine (by default, the files are in `C:\Program Files\CodeGear\RAD Studio\5.0\Bin`):

- rmtdbg105.exe
- bccide.dll
- bordbk105.dll

- bordbk105N.dll
- comp32x.dll
- dbkpro100.dll
- DCC100.DLL

3. Copy the files from your local machine to the directory you created on the remote machine.
4. On the remote computer, register `bordbk105.dll` and `bordbk105n.dll`by running the `regsvr32.exe` registration utility. For example, on Windows XP, enter `C:\Windows\System32\regsvr32.exe bordbk105.dll` at the command prompt, then enter `C:\Windows\System32\regsvr32.exe bordbk105n.dll`.
5. If you are debugging an ASP.NET application, copy Borland.dbkasp.dll to the Install\GlobalAssemblyCache directory on the remote machine. If you are debugging an ASP.NET application, register the Borland.dbkasp.dll in the GlobalAssemblyCache using the Microsoft .NET gacutil.exe utility. For example, on Windows XP with Microsoft .NET Framework SDK, enter `C:\Program Files\Microsoft.NET\SDK\v1.1\Bin\gacutil Borland.dbkasp.dll`.

See Also

Overview of Remote Debugging (see page 12)

Establishing a Connection for Remote Debugging (see page 127)

Preparing Files for Remote Debugging (see page 128)

2.2.14 Establishing a Connection for Remote Debugging

You must establish a TCP/IP connection between the local and remote machines in preparation for remote debugging. This connection uses multiple ports that are chosen dynamically by Windows. The remote debug server listens on one port, and a separate port is opened for each application that is being debugged. A firewall that only allows connections to the listening port will prevent the remote debugger from working.

Note: If the remote machine uses the firewall included with Windows XP service pack 2, you will receive a message asking whether CodeGear remote debugging service should be allowed. You must indicate that this is allowed.

Warning: The connection between RAD Studio and the remote debug server is a simple TCP/IP socket, with neither encryption nor authentication support. Therefore, the remote debug server should not be run on a computer that can be accessed over the network by untrusted clients.

To connect the local machine and the remote machine

1. Ensure that the remote debugger is installed on the remote machine.
2. Ensure that the executable files and symbol files (.tds. .rsm and .pdb) have been copied to the remote machine.
3. On the remote machine, start rmtdbg105.exe with the **-listen** argument. `rmtdbg105.exe -listen` This starts the remote debugger's listener and directs it to wait for a connection from your host machine's IDE.
4. On the local machine, choose **Run ▶ Attach to Process**. This displays the **Attach to Process** dialog.
5. Specify the host name or TCP/IP address for the remote machine, then click **Refresh**. A list of processes running on the remote machine is displayed. This verifies the connectivity between the local and remote machines.
6. On the local machine, choose **Run ▶ Load Process ▶ Remote**. This displays the **Remote** page of the **Load Process** dialog.
7. In the **Remote path** field, specify the full path for the directory on the remote machine into which you copied the executable files and symbol files. The name of the executable must be included. For example, if you are debugging a program1.exe, and you copy this to a directory named RemoteDebugFiles\Program1 on the remote machine, specify `C:\RemoteDebugFiles\Program1\program1.exe`.
8. In the **Remote host** field, specify the host name or TCP/IP address for the remote machine.

9. Click the **Load** button. This connects the IDE on the local machine to the debugger on the remote machine.

Once this connection is established, you can use the IDE on the local machine to debug the application as it runs on the remote machine.

Note: You cannot interact directly with the remote application through the remote debugger. For interactive debugging, you can establish a remote desktop connection.

See Also

Overview of Remote Debugging (↗ see page 12)

Installing a Debugger on a Remote Machine (↗ see page 126)

Preparing Files for Remote Debugging (↗ see page 128)

Debugger (↗ see page 836)

Linker (↗ see page 840)

2.2.15 Preparing Files for Remote Debugging

Executable files and symbol files must be copied to the remote machine after they are compiled. You must set the correct options on your local machine in order to generate these files.

To prepare files for debugging on a remote machine

1. Open the project on your local machine.
2. For Delphi, choose **Project**▶**Options**▶**Linker** and verify that the **Include remote debug symbols** option is checked. This directs the compiler to generate a symbol file. The following extensions are used in symbol files (for Delphi projects):

Language	Debug symbol file extension
Delphi for Win32	.rsm
Delphi for .NET	.rsm and .pdb
C++	.tds
C#	.pdb

3. Compile the project on your local machine.
4. Copy the executable files and symbol files for the project to the remote machine.
5. Choose **Run**▶**Load Process**
6. Specify the directory into which you copied the symbol files in the **Debug symbols search path** field.
7. Click **OK**.

See Also

Overview of Remote Debugging (↗ see page 12)

Installing a Debugger on a Remote Machine (↗ see page 126)

Establishing a Connection for Remote Debugging (↗ see page 127)

Setting the Search Order for Debug Symbol Tables (↗ see page 129)

Debugger Options (↗ see page 836)

Linker (↗ see page 840)

Symbol Tables (see page 845)

2.2.16 Setting the Search Order for Debug Symbol Tables

Symbol tables are used internally during debugging. By default, RAD Studio locates and uses all symbol tables available. However, you can control the order in which these symbol tables are searched. You can also limit the search to specific symbol tables, which can speed up the debugging process.

The extensions for symbol table files vary by personality.

- Delphi Win32, does not use external symbol files because the compiler holds the symbols tables in memory. However, if you are debugging a remote application, you must generate symbol files with the .RSM extension.
- Delphi.NET, VB.NET and C# symbol files use the .PDB extension.
- C++ symbol files use the .TDS extension. However, if debug information is contained in the PE file, external symbol tables are not used.

To set the order in which symbol tables are searched

1. Specify the general project search path.
2. Specify the global path for all projects.
3. Specify the language-specific path for the project.
4. Specify the language-specific global path.

To specify the general project search path

1. Choose **Project**▶**Options**▶**Debugger**▶**Symbol Tables**.
2. In the **Debug symbols search path** field, type or navigate to the path to the symbols table that you want the debugger to use.

   Note: If you want to limit the search to specific symbol tables, proceed to the next step. If you want the debugger to search all paths, click OK

   to finish specifying the general project search path.
3. Uncheck the **Load all symbols** check box.
4. Click **New**. The **Add Symbol Table Search Path** dialog displays.
5. Enter the name of the module you are debugging and one or more paths that contain the symbol table for that module. If you specify multiple paths, use a semicolon to separate them.
6. Click **OK**. The **Add Symbol Table Search Path** dialog closes and the module and path you added are displayed in the table.

   Note: You can use this list to specify modules and paths that the debugger is to avoid searching by using a blank path and checking the Load symbols for unspecified modules

   check box.
7. Click **OK**.

To specify the global path for all projects (for Delphi and C++ only)

1. Choose **Tools**▶**Options**▶**Debugger Options**▶**CodeGear Debuggers**.
2. In the **Debug symbols search path** field, type or navigate to the path to the symbols table that you want the debugger to use.
3. Click **OK**.

To specify the language-specific path for the project

1. Choose **Project**▶**Options**▶**Directories/Conditionals** . The **Directories/Conditionals** page contains four fields in which you

can specify a path for Win32 and .NET symbol tables. They are searched in the following order during debugging:

1. **Search path**
2. **Package output directory**
3. **DCP output directory**
4. **Output directory**

2. In each of these fields, type or navigate to the path to the symbols table that you want the debugger to use.
3. Click **OK**.

To specify global paths

1. Choose **Tools**▶**Options**▶**Delphi Options**▶**Library (Win32 or NET)**. Depending on the language, the **Library** page contains two or three fields in which you can specify a path for Win32 and .NET symbol tables. They are searched in the following order during debugging:

1. **Browsing path**
2. **DCP output directory** (not used for C++)
3. **Package output directory**

2. In each of these fields, type or navigate to the path to the symbols table that you want the debugger to use.
3. Click **OK**.

See Also

Overview of Debugging (see page 10)

Overview of Remote Debugging (see page 12)

Symbol Tables (see page 845)

Add Symbol Table Search Path (see page 826)

CodeGear Debuggers (see page 986)

Directories/Conditionals (see page 838)

Library (see page 991)

2.2.17 Resolving Internal Errors

The error message, `Internal Error: X1234` indicates that the compiler has encountered a condition, other than a syntax error, that it cannot successfully process.

Tip: Internal error numbers indicate the file and line number in the compiler where the error occurred. This information may help Technical Support services track down the problem. Be sure to record this information and include it with your internal error description.

To resolve an internal error

1. If the error occurs immediately after you have modified code in the editor, go back to the place where you made your changes and make a note of what was changed.
2. If you can undo or comment out the change and then recompile your application successfully, it is possible that the programming construct that you introduced exposed a problem with the compiler. If so, follow the procedure on reviewing code below.

If the problem still exists

1. Delete all of the `.dcuil` files associated with your project.
2. Close your project completely using **File ▶ Close All**.
3. Reopen your project. This will clear the unit cache maintained in the IDE. Alternatively, you can close the IDE and restart.
4. Another option is to try and recompile your application using the **Project ▶ Build** option so that the compiler will regenerate all of your dcuils.
5. If the error is still present, exit the IDE and try to compile your application using the command line version of the compiler (`dccil.exe`) from a command prompt. This will remove the unit caching of the IDE from the picture and could help to resolve the problem.

Review your code at the last modification point

1. If the problem still exists, go back to the place where you last made modifications to your file and review the code. Typically, most internal errors can be reproduced with only a few lines of code and frequently the code involves syntax or constructs that are rather unusual or unexpected. If this is the case, try modifying the code to do the same thing in a different way. For example, if you are typecasting a value, try declaring a variable of the cast type and do an assignment first.

```
begin
    if Integer(b) = 100 then...
end;
var
 a: Integer;
begin
  a := b;
  if a = 100 then...
end;
```

Here is an example of unexpected code that you can correct to resolve the error:

```
var
    A : Integer;
begin
 { Below the second cast of A to Int64 is unnecessary; removing it can avoid the Internal
Error. }
 if Int64(Int64(A))=0 then
end;
```

2. In this case, the second cast of `A` to an `Int64` is unnecessary and removing it corrects the error. If the problem seems to be a `while...do` loop, try using a `for...do` loop instead. Although this does not actually solve the problem, it may help you to continue work on your application. If this resolves the problem, it does not mean that either `while` loops or `for` loops are broken but more likely it means that the manner in which you wrote your code was unexpected.
3. Once you have identified the problem, we ask that you create the smallest possible test case that still reproduces the error and submit it to Borland.

Other techniques for resolving internal errors

1. If error seems to be on code contained within a `while...do` loop try using a `for...do` loop instead or vice versa.
2. If it uses a nested function or procedure (a procedure/function contained within a procedure/function) try unnesting them.
3. If it occurs on a typecast look for alternatives to typecasting like using a local variable of the type you need.
4. If the problem occurs within a with statement try removing the with statement altogether.
5. Try turning off compiler optimizations under Project **Options ▶ Compiler**.

When all else fails

1. Typically, there are many different ways to write any single piece of code. You can try and resolve an internal error by changing the code. While this may not be the best solution, it may help you to continue to work on your application. If this resolves the problem, it does not mean that either `while` loops or `for` loops are broken but perhaps that the manner in which you have written your code was unexpected and therefore resulted in an error.

2. If you've tried your code on the latest release of the compiler and it is still reproducible, create the smallest possible test case that will still reproduce the error and submit it to CodeGear. If it is not reproducible on the latest version, it is likely that the problem has already been fixed.

Configuring the IDE to avoid internal errors

1. Create a single directory where all of your `.dcpil` files (precompiled package files) are placed. For example, create a directory called `C:\DCPIL` and under Tools **Environment Options** select the **Library** tab and set the DCPIL output directory to `C:\DCPIL`. This setting will help ensure that the `.dcpil` files the compiler generates are always up-to-date. This is useful when you move a package from one directory to another. You can create a `.dcuil` directory on a per-project basis using Project**Options** ▶ **Directories/Conditionals** ▶ **Unit** output directory.
2. The key is to use the most up-to-date versions of your `.dcuil` and `.dcpil` files. Otherwise, you may encounter internal errors that are easily avoidable.

See Also

Runtime errors (↗ see page 509)

Fatal errors (↗ see page 511)

I/O errors (↗ see page 510)

Operating system errors (↗ see page 512)

List of all Delphi compiler errors and messages (↗ see page 311)

2.3 Deploying Applications

This section provides how-to information on deploying applications. There are two ways to deploy an application — manually, or using the Deployment Manager. Currently, the Deployment manager is only for use with ASP.NET applications. Manual procedures for deploying applications are described in the appropriate Win32 area in this help system (for example, Delphi, ECO, or COM).

Topics

Name	Description
Deploying ASP.NET applications (↗ see page 133)	The Deployment Manager can be used with ASP.NET applications to collect all of the .aspx, asax, Web.config and other assembly files, as well as related assemblies.
Deploying the AdoDbx Client (↗ see page 134)	You can deploy AdoDbx Client applications in several ways.

2.3.1 Deploying ASP.NET applications

The Deployment Manager can be used with ASP.NET applications to collect all of the .aspx, asax, Web.config and other assembly files, as well as related assemblies.

To deploy an ASP.NET application

1. Open the ASP.NET project that you want to deploy. In the **Project Manager** window, right-click the **Deployment** option under the library that corresponds to your project name.
2. Select New ASP.NET Deployment. This opens the **Deployment Manager** window. There are three main categories of files: ASP.NET Markup Files, Executables, and Config Files.
3. The strong name assemblies Borland.Data.Common.dll and Borland.Data.Provider.dll have traditionally existed in the Global Assembly Cache (GAC). Add these two files to the project directory by right-clicking **References** under the library that corresponds to your project name and then selecting **Add Reference**. Click each of the file names and then click **Add Reference**. After both references appear in the lower section of the **Add Reference** window, choose the **OK** button.
4. Open the **References** node of the library tree, and the new .dll files are listed. Select each of the .dll files that you added in the previous step, and set the **Copy Local Assembly Property** in the **Project Manager** to True.
5. Right-click **References** again to add your database-specific .dll file(s) to the assembly. For instance, if you are running MySQL Server, you would add the Borland.Data.Mysql driver and set the Copy Local Assembly Property in the **Project Manager** to True.
6. Recompile your project.
7. Highlight your deployment window again under the **Deployment** option of the Project Manager window.
8. Browse or enter a path to the target machine, where you want to deploy your application. Then click the icon next to each file name. This moves each file to the 'Destination' side of the window. When all of your source files appear on the 'Destination' side of the window, they have been deployed to the target computer.

See Also

Deploying Applications (↗ see page 19)

Using the Deployment Manager

2.3.2 Deploying the AdoDbx Client

You can deploy AdoDbx Client applications in several ways.

To deploy updating machine.config

1. Register the following files in the Global Assembly Cache (GAC) or have them in the same directory as the project .exe:

- Borland.Data.AdoDbxClient.dll
- Borland.Data.DbxCommonDriver.dll
- Borland.Data.DbxDynalinkDriver.dll
- Borland.Data.DbxReadOnlyMetaData.dll
- Borland.Delphi.dll
- Borland.VclDbRtl.dll

2. The Microsoft .NET machine.config file must have an entry added for the provider. Locate the machine.config file in the Windows\Microsoft.NET directory and add the following entry to the `<DbProviderFactories>` section:

```
<add name="AdoDbx Data Provider"
invariant="Borland.Data.AdoDbxClient"
description=".Net Framework Data Provider for dbExpress Drivers"
type="Borland.Data.TAdoDbxProviderFactory, Borland.Data.AdoDbxClient,
Version=11.0.5000.0,Culture=neutral,PublicKeyToken=91d62ebb5b0d1b1b"/>
```

3. The machine.config file must have an entry added for the connection string. Add an entry to the `<connectionStrings>` section similar to this one for an Interbase connection:

```
<add name="IBConnection"
connectionString="SomeName=IBCONNECTION;
drivername=Interbase;
database=someDatabasePathName\database.gdb;
rolename=RoleName;user_name=UserName;password=password;
sqldialect=3;localecode=0000;blobsize=-1;commitretain=False;waitonlocks=True;
interbase transisolation=ReadCommited;
trim char=False"
providerName="Borland.Data.AdoDbxClient"/>
```

To deploy without updating machine.config

1. An application can also deploy without updating the machine.config by directly using the **TAdoDbxProviderFactory** class in the **Borland.Data.AdoDbxClientProvider** unit.
2. Register the following files in the Global Assembly Cache (GAC) or have them in the same directory as the project .exe:

- Borland.Data.AdoDbxClient.dll
- Borland.Data.DbxCommonDriver.dll
- Borland.Data.DbxDynalinkDriver.dll
- Borland.Data.DbxReadOnlyMetaData.dll
- Borland.Delphi.dll
- Borland.VclDbRtl.dll

3. Your project needs to reference **Borland.Data.AdoDbxClient.dll**. Click **Project ▶ Add Reference**. In the **Add Reference** dialog, click **Browse** and select the file **Borland.Data.AdoDbxClient.dll**, which is in the **Program Files\CodeGear\RAD Studio\5.0\bin** directory.
4. Create a <projectname>.exe.config file. For example, if the project is named Project1, the file name is Project1.exe.config. Add the following text to the file:

```
<configuration>
    <system.data>
        <DbProviderFactories>
            <add name="AdoDbx Data Provider"
invariant="Borland.Data.AdoDbxClient"
description=".Net Framework Data Provider for dbExpress Drivers"
type="Borland.Data.TAdoDbxProviderFactory, Borland.Data.AdoDbxClient, Version=11.0.5000.0,
Culture=neutral, PublicKeyToken=a91a7c5705831a4f"/>
        </DbProviderFactories>
    </system.data>
    <connectionStrings>
        <add name="IBConnection"
connectionString="SomeName=IBCONNECTION;
drivername=Interbase;
database=someDatabasePathName\database.gdb;
rolename=RoleName;user_name=UserName;password=password;
sqldialect=3;localecode=0000;blobsize=-1;commitretain=False;waitonlocks=True;
interbase transisolation=ReadCommited;
trim char=False"
providerName="Borland.Data.AdoDbxClient"/>
    </connectionStrings>
</configuration>
```

Note: Although this approach makes deployment easier, the TAdoDbxProviderFactory

can only create ADO.NET 2.0 objects for the AdoDbx Client provider.

See Also

AdoDbx Client Overview

2.4 Editing Code Procedures

This section provides how-to information on using the features of the **Code Editor**.

Topics

Name	Description
Using Code Folding (see page 137)	Code folding lets you collapse (hide) and expand (show) your code to make it easier to navigate and read. RAD Studio generates code that contains code folding regions, but you can add your own regions as needed.
Creating Live Templates (see page 138)	While using the **Code Editor**, you can add your favorite code constructs to the **Template Manager** to create a library of the templates you use most often.
Creating Template Libraries (see page 138)	Template libraries are custom project templates that specify how a project should look and what it should contain. When you create a custom template library, it is placed in the **New Files** dialog box where is accessible for creating a project using **File ▶ New**. You can create template library projects from scratch, or you can use projects previously created by you or other developers as the basis for template libraries. To use an existing project, you simply create an XML file with the extension `.bdstemplatelib` which describes the project and is used to create the template library using that... more (see page 138)
Customizing Code Editor (see page 141)	CodeGear RAD Studio lets you customize your **Code Editor** by using the available settings to modify keystroke mappings, fonts, margin widths, colors, syntax highlighting, and indentation styles.
Finding References (see page 141)	The Find References refactoring feature helps you locate any connections between a file containing a symbol you intend to rename and other files where that symbol also appears. A preview allows you to decide how you want the refactoring to operate on specific targets or on the group of references as a whole.
Finding Units and Using Namespaces (Delphi, C#) (see page 142)	Depending on which language you are using, you can use a refactoring feature to locate namespaces or units. If you are using C#, you can use the Use Namespace command to import namespaces into your code files, based on an object in your code. If you are using Delphi, you can use the Find Unit command to locate and add units to your code file based on objects in your code. The **Use Namespace** dialog appears when you select a C# object name and select the **Use Namespace** command. The import operation attempts to identify and display the most likely... more (see page 142)
Recording a Keystroke Macro (see page 142)	You can record a series of keystrokes as a macro while editing code. After you record a macro, you can play it back to repeat the keystrokes during the current IDE session.
Refactoring Code (see page 143)	Refactoring refers to the capability to make structural changes to your code without changing the functionality of the code. Code can often be made more compact, more readable, and more efficient through selective refactoring operations. RAD Studio provides a set of refactoring operations that can help you re-architect your code in the most effective and efficient manner possible. Refactoring operations are available for Delphi, C#, and C++. However, the refactorings for C# and C++ are limited in number. You can access the refactoring commands from the Refactoring menu or from a right-click context menu while in the **Code Editor**.... more (see page 143)
Using Bookmarks (see page 145)	You can mark a location in your code with a bookmark and jump directly to it from anywhere in the file. You can set up to ten bookmarks. Bookmarks are preserved when you save the file and available when you reopen the file in the **Code Editor**.
Using Class Completion (see page 145)	Class completion automates the definition of new classes by generating skeleton code for Delphi class members that you declare.
Using Code Insight (see page 146)	**Code Insight** (sometimes referred to as **Code Completion**) is a set of features in the **Code Editor** and the HTML **Tag Editor** that provide code completion, display code parameter lists, and show tool tips for expressions and symbols. The hint window list box filters out all interface method declarations that are referred to by property read or write clauses. The list box displays only properties and stand-alone methods declared in the interface type.

Using Live Templates (see page 148)	Live templates are reusable code statements that are accessible from the **Code Editor**. You can insert pre-defined code segments into your code or add your own code snippets to the **Template** window. **Note:** If a template has one or more jump points that are editable, it will automatically enter **SyncEdit** mode when you are inserting it into your code. The jump points allow you to navigate between different areas of the template, using the Tab key and `SHIFT+Tab` keys. Pressing `ESC`, `Enter`,(or pressing the`Tab` key) from the last jump point exits **SyncEdit** mode and puts the... more (see page 148)
Using the History Manager (see page 149)	The **History Manager** lets you view and compare versions of a file, including multiple backup versions, saved local changes, and the edit buffer of unsaved changes. For simplicity, the following procedures uses a small text file to introduce the functionality of the **History Manager**. However, the **History Manager** is available for most files, including source code and HTML files.
Using Sync Edit (see page 150)	The Sync Edit feature lets you simultaneously edit indentical identifiers in selected code. For example, in a procedure that contains three occurrences of `label1`, you can edit just the first occurrence and all the other occurrences will change automatically.

2.4.1 Using Code Folding

Code folding lets you collapse (hide) and expand (show) your code to make it easier to navigate and read. RAD Studio generates code that contains code folding regions, but you can add your own regions as needed.

To collapse and expand code

1. In the **Code Editor**, click the minus (-) sign to the left of a code block to collapse the code.
2. Click the plus (+) sign to expand the code block.

   Tip: To turn off code folding for the current edit session, press and hold Ctrl+Shift

   , and then `K`, and then `O`. To collapse the nearest code block, press and hold `Ctrl+Shift`, and then `K`, and `E`. To expand the nearest code block, press and hold `Ctrl+Shift`, and then `K`, and `U`. To expand all code, press and hold `Ctrl+Shift` and then press `K`, and `A`.

To add a code folding region

1. In the **Code Editor**, use the following preprocessor directives to surround a block of code:

```
{$region 'Optional text that appears when the code block is folded'}
.
.
.
{$endregion}
#region Optional text that appears when the code block is folded
.
.
.
#endregion
#pragma region optional text
.
.
.
#pragma end_region
```

The region is marked with a minus (-) sign.

2. Click the minus sign (-) to collapse the region.

See Also

Customizing Code Editor (see page 141)

Using Code Insight (see page 146)

2.4.2 Creating Live Templates

While using the **Code Editor**, you can add your favorite code constructs to the **Template Manager** to create a library of the templates you use most often.

To add a Live Template using the Menu Commands

1. While you are working in the **Code Editor**, choose **File**▶**New**▶**Other**▶**Other Files** and then select the Live Template icon.
2. Fill in the template name, description, author, and code language attributes. Then type in the code for your template between the `<![CDATA[]]>` tag and the `</code>` tag.

   Note: The Name

   and `Language` fields in the template are required.
3. Choose the Save command from the **File** pull-down menu in the **Code Editor** (or type `CTRL + S`). Your new template now appears in the IDE tree of the **Template Manager** window. It is saved, by default, into the `\5.0\Objrepos\code_templates\` directory.

To add a Live Template using the Template Manager window

1. In the **Code Editor**, choose **View**▶**Templates**.
2. In the **Template Manager** window, click the **New** button. This will put an XML outline for a code template in the **Code Editor** main window. You can also select code in the editor before you click the **New** button.
3. Fill in the template name, description, author, and code language attributes. Then type in the code for your template between the `<![CDATA[]]>` tag and the `</code>` tag.

   Note: The Name

   and `Language` fields in the template are required.
4. Choose the Save command from the **File** pull-down menu in the **Code Editor** (or type `CTRL + S`). Your new template now appears in the IDE tree of the **Template Manager** window. It is saved, by default, in the `\5.0\Objrepos\code_templates\` directory.

See Also

Using Live Templates (see page 148)

Customizing the Code Editor (see page 141)

2.4.3 Creating Template Libraries

Template libraries are custom project templates that specify how a project should look and what it should contain. When you create a custom template library, it is placed in the **New Files** dialog box where is accessible for creating a project using **File**▶**New**.

You can create template library projects from scratch, or you can use projects previously created by you or other developers as the basis for template libraries. To use an existing project, you simply create an XML file with the extension `.bdstemplatelib` which describes the project and is used to create the template library using that project.

Note: When creating a project to use with a template library, the project should be located in a subdirectory that contains no other projects. Also, all of the files that are in the project should be located within the subdirectory or its child subdirectories.

To create a Template Library

1. Create a new project or open an existing project which will be the basis for the custom template library. Make any modifications to the project to customize it for the template library.
2. Save and close the project.
3. Choose **File ▶ New ▶ Other ▶ WebDocuments** and double-click the **XML File** icon.
4. Replace the default contents of the new XML file with the following sample content:

```
<TemplateLibrary Version="1.0" id="">
  <Name><Name/>
  <Description><Description/>
  <Items>
   <Item id="" Creator="">
      <Name>Name of template library here <Name/>
      <Description>Custom Project Template<Description/>
   <Author><Author/>
   <Icon>MyTemplate\MyTemplateIcon.ico<Icon/>
        <Projectfile>MyTemplate.dproj</Projectfile>
       <DefaultProjectName>MyTemplate<DefaultProjectName/>
    <FilePath>MyTemplate<FilePath/>
    </Item>
  </Items>
</TemplateLibrary>
```

Important:

- The `id=""` attribute of the `<TemplateLibrary>` element should be something unique to avoid conflicts with other template libraries. A good practice is to include your name or the name of your company as part of the `id`.
- The `Creator=""` attribute in the `<Item>` element specifies which page of the **New Items** dialog box displays the icon for this template library. You can put the project icon on a specific page for the type of project it creates. Below are the possible `Creator=""` attribute values:

Project Type	Item Creator Attribute Value
C++ Projects	`Creator="CBuilderProjectRepositoryCreator"`
Delphi Projects	`Creator="DelphiProjectRepositoryCreator"`
Delphi for .NET Projects	`Creator="DelphiDotNetProjectRepositoryCreator"`
Delphi for ASP .NET Projects	`Creator="AspDelphiProjectRepositoryCreator"`

5. Save the `.bdstemplatelib.xml` file to a directory above the project directory.

   Note: The `<FilePath>` element in the `.bdstemplatelib` file indicates the location of the project directory relative to the location of the `.bdstemplatelib` file.

6. Edit the `.bdstemplatelib.xml` content to customize it for your own template library:

- Add the template library name, description, and `Creator` attribute value.
- Edit the project name, project path information, icon path, and file name. Optionally, you can add your name as author.
- Specify the relative path to the `.bdstemplatelib.xml` in the `<FilePath>` value. For example, if your project is in `C:\MyProjects\TemplateLibraries\MyTemplate`, and you put the XML file in `C:\MyProjects\TemplateLibraries`, the `<FilePath>` value in the XML file would be `<FilePath>TemplateLibraries\MyTemplate</FilePath>`.

7. Choose **Tools ▶ Template ▶ Libraries** to open the **Template Libraries** dialog box.
8. Click the **Add** button, browse to and select the `.bdstemplatelib.xml` file you just created, and click **Open**. The new template library is added to the list in the **Template Libraries** dialog box. It is also added to the specified page of the **New Files** dialog box. Click **OK** to close the **Template Libraries** dialog box.

To use this template library for creating a new project, choose **File ▶ New ▶ Other**, and select your template library in the **New files** dialog box.

Example

```
<TemplateLibrary Version="1.0" id="CompanyXYZASPWebSiteProject">
  <Name>ASPWebSiteProject<Name/>
  <Description>ASP.NET WebSite Project Template<Description/>
  <Items>
   <Item id="ASPWebSiteProject" Creator="AspDelphiProjectRepositoryCreator">
      <Name>ASP.NET WebSite Project<Name/>
      <Description>ASP.NET WebSite<Description/>
   <Author>John Smith<Author/>
   <Icon>ASPWebsiteProject\ASPWebsiteProjectIcon.ico<Icon/>
        <Projectfile>ASPWebSiteProject.dproj</Projectfile>
       <DefaultProjectName>ASPWebSiteProject<DefaultProjectName/>
    <FilePath>ASPWebSiteProject<FilePath/>
    </Item>
  </Items>
</TemplateLibrary>
```

If you have several related projects, you can use a single `.bdstemplatelib` template library file to list all the projects.

To combine multiple projects in one template library file

1. Put all the project folders at the same level in the same project sub-folder.
2. Create the `.bdstemplatelib` template library file at the level above the folder containing all the projects.
3. Add the content for the first project as described above.
4. Add an additional `<Item></Item>` to the `<Items></Items>` element for each project in the group, giving each `<Item></Item>` a unique `id="` attribute.

Example

```
<TemplateLibrary Version="1.0" id="CodeGearASPWebSiteProject">
  <Name>ASPWebSiteProject<Name/>
  <Description>ASP.NET WebSite Project Template<Description/>
  <Items>
    <Item id="ASPWebSiteProject" Creator="AspDelphiProjectRepositoryCreator">
      <Name>ASP.NET WebSite Project<Name/>
      <Description>ASP.NET WebSite Project<Description/>
      <Author>CodeGear<Author/>
      <Icon>ASPWebsiteProject\ASPWebsiteProjectIcon.ico<Icon/>
      <Projectfile>ASPWebSiteProject.dproj</Projectfile>
      <DefaultProjectName>ASPWebSiteProject<DefaultProjectName/>
      <FilePath>ASPWebSiteProject<FilePath/>
    </Item>
    <Item id="ASPWebSiteProjectMP" Creator="AspDelphiProjectRepositoryCreator">
       <Name>ASP.NET WebSite Project Master Page<Name/>
       <Description>ASP.NET WebSite Project Master Page<Description/>
      <Author>CodeGear<Author/>
      <Icon>ASPWebsiteProjectMP\ASPWebsiteProjectMPIcon.ico<Icon/>
       <Projectfile>ASPWebSiteProjectMP.dproj</Projectfile>
       <DefaultProjectName>ASPWebSiteProjectMP<DefaultProjectName/>
      <FilePath>ASPWebSiteProjectMP<FilePath/>
    </Item>
    <Item id="ASPWebSiteProjectForm" Creator="AspDelphiProjectRepositoryCreator">
       <Name>ASP.NET WebSite Project Information Form<Name/>
      <Description>ASP.NET WebSite Information Form<Description/>
      <Author>CodeGear<Author/>
      <Icon>ASPWebsiteProjectForm\ASPWebsiteProjectFormIcon.ico<Icon/>
       <Projectfile>ASPWebSiteProjectForm.dproj</Projectfile>
      <DefaultProjectName>ASPWebSiteProjectForm<DefaultProjectName/>
      <FilePath>ASPWebSiteProjectForm<FilePath/>
    </Item>
  </Items>
```

```
</TemplateLibrary>
```

See Also

Overview of Template Libraries (see page 50)

2.4.4 Customizing Code Editor

CodeGear RAD Studio lets you customize your **Code Editor** by using the available settings to modify keystroke mappings, fonts, margin widths, colors, syntax highlighting, and indentation styles.

To customize general Code Editor options

1. Choose **Tools** ▶ **Options**.
2. Click **Editor Options**.
3. Select any of the customization options and make modifications.
4. Click **OK** to apply the modifications to the **Code Editor**.

See Also

Using Code Folding (see page 137)

Using Code Insight (see page 146)

2.4.5 Finding References

The Find References refactoring feature helps you locate any connections between a file containing a symbol you intend to rename and other files where that symbol also appears. A preview allows you to decide how you want the refactoring to operate on specific targets or on the group of references as a whole.

To create a Find References list

1. Open a project.
2. Select an identifier in the **Code Editor**.
3. Choose **Search** ▶ **Find References**.

   Note: You can also invoke Find References with the keyboard shortcut Shift+Ctrl+Enter

   .
4. Double-click a node in the window to go to that location in the **Code Editor**.

   Note: If you continue to perform Find References operations without clearing the results, the new results are appended in chronological order to the existing results in the window.

To clear results from the Find References window

1. Select a single reference or a node.

   Note: No matter which you select, you get the same results. The entire node will be cleared.

2. Click the **Refactor Delete** icon ✖ at the top of the **Find References** window, to delete the selected item and any item in that

result set.

Note: Deleting items from the Find References

window does not delete them from your actual code files or your project.

To clear all results from the Find References window

1. Select any item in the window.
2. Click the **Remove All References** icon at the top of the **Find References** window. This action clears all results from the window.

   Note: Deleting items from the Find References

   window does not delete them from your actual code files or your project.

See Also

Refactoring Overview (see page 57)

Find References Overview (see page 65)

2.4.6 Finding Units and Using Namespaces (Delphi, C#)

Depending on which language you are using, you can use a refactoring feature to locate namespaces or units. If you are using C#, you can use the Use Namespace command to import namespaces into your code files, based on an object in your code. If you are using Delphi, you can use the Find Unit command to locate and add units to your code file based on objects in your code.

The **Use Namespace** dialog appears when you select a C# object name and select the **Use Namespace** command. The import operation attempts to identify and display the most likely namespaces. You can select multiple namespaces to add to the using clause. The feature works identically in Delphi, although in a Delphi project, the operation attempts to find the appropriate unit containing the definition of the selected object, then adds the selected unit to the uses clause.

See Also

Refactoring Overview (see page 57)

Refactoring Code (see page 143)

2.4.7 Recording a Keystroke Macro

You can record a series of keystrokes as a macro while editing code. After you record a macro, you can play it back to repeat the keystrokes during the current IDE session.

To record a macro

1. In the **Code Editor**, click the record macro button ● at the bottom of the code window to begin recording.
2. Type the keystrokes that you want to record.
3. When you have finished typing the keystroke sequence, click the stop recording button 🟩.
4. To record another macro, repeat the previous steps.

   Note: Recording a macro replaces the previously recorded macro.

The macro is now available to use during the current IDE session.

To run a macro

1. In the **Code Editor**, position the cursor in the code where you want to run the macro.
2. Click the macro playback button ▶ to run the macro. If the button is dimmed, no macro is available.

2.4.8 Refactoring Code

Refactoring refers to the capability to make structural changes to your code without changing the functionality of the code. Code can often be made more compact, more readable, and more efficient through selective refactoring operations. RAD Studio provides a set of refactoring operations that can help you re-architect your code in the most effective and efficient manner possible.

Refactoring operations are available for Delphi, C#, and C++. However, the refactorings for C# and C++ are limited in number. You can access the refactoring commands from the Refactoring menu or from a right-click context menu while in the **Code Editor**.

The **Undo** capability is available for all refactoring operations. Some operations can be undone using the standard **Undo** (`CTRL+Z`) menu command, while the rename refactorings provide a specific Undo feature.

To rename a symbol

1. In the **Code Editor**, click the identifier to be renamed. The identifier can be a method, variable, field, class, record, struct, interface, type, or parameter name.
2. From either the main menu or the **Code Editor** context menu, choose **Refactor**▶**Rename**.
3. In the **Rename** dialog box, enter the new identifier in the **New Name** field.
4. Leave **View references before refactoring** checked. If this option is unchecked, the refactoring is applied immediately, without a preview of the changes.
5. Click **OK**. The **Refactorings** dialog box displays every occurrence of the identifier to be changed.
6. Review the proposed changes in the **Refactorings** dialog box and use the **Refactor** button at the top of the dialog box to perform all of the refactorings listed. Use the **Remove Refactoring** button to remove the selected refactoring from the dialog box.

To declare a variable

1. In the **Code Editor**, click anywhere in a variable name that has not yet been declared.

   Note: Any undeclared variable will be highlighted with a red wavy underline by Error Insight.

2. From either the main menu or the **Code Editor** context menu, choose **Refactor**▶**Declare Variable**. If the variable has already been declared in the same scope, the command is not available.
3. Fill in the **Declare New Variable** dialog box as needed.
4. Click **OK**.

The variable declaration is added to the procedure, based on the values you entered in the **Declare New Variable** dialog box.

To declare a field

1. In the **Code Editor**, click anywhere in a field name that has not yet been declared.
2. From either the main menu or the **Code Editor** context menu, choose **Refactor**▶**Declare Field**.
3. Fill in the **Declare New Field** dialog box as needed.

4. Click **OK**.

The new field declaration is added to the type section of your code, based on the values you entered in the **Declare New Field** dialog box.

Note: If the new field conflicts with an existing field in the same scope, the Refactorings

dialog box is displayed, prompting you to correct the conflict before continuing.

To create a method from a code fragment

1. In the **Code Editor**, select the code fragment to be extracted to a method.
2. From either the main menu or the **Code Editor** context menu, choose Refactor ▶ Extract Method. The **Extract Method** dialog box is displayed.
3. Enter a name for the method in the **New method name** field, or accept the suggested name.
4. Review the code in the **Sample extracted code** window.
5. Click **OK**.

RAD Studio moves the extracted code outside of the current method, determines the needed parameters, generates local variables if necessary, determines the return type, and replaces the original code fragment with a call to the new method.

To convert a string constant to a resource string (for the Delphi language only)

1. In the **Code Editor**, select the quoted string to be converted to a resource string, for example, in the following code, insert the cursor into the constant `Hello World`:

```
procedure foo;
begin
    writeLn('Hello World');
end;
```

2. From either the main menu or the **Code Editor** context menu, choose Refactor ▶ Extract Resource String.

Note: You can also use the Shift+Ctrl+L

keyboard shortcut. The **Extract Resource String** dialog box is displayed.

3. Enter a name for the resource string or accept the suggested name (the `Str`, followed by the string).
4. Click **OK**.

The `resourcestring` keyword and the resource string are added to the implementation section of your code, and the original string is replaced with the new resource string name.

```
resourcestring
    strHelloWorld = 'Hello World';

procedure foo;
begin
    writeLn(StrHelloWorld);
end.
```

To find and add a namespace or unit to the uses clause

1. In the **Code Editor**, click anywhere in a the variable name whose unit you want to add to the `uses` clause (Delphi) or the namespace you want to add to the `using` clause (C#).
2. From either the main menu or the **Code Editor** context menu, choose Refactor ▶ Find Unit. The **Find Unit** dialog box displays a selection list of applicable Delphi units.

Note: If you are coding in C#, the dialog box is called the Use Namespace

dialog box.

3. Select the unit or namespace that you want to add to the `uses` or `using` clause in the current scope. You can select as many units or namespaces as you want.
4. If you are coding in Delphi, choose where to insert the reference, either in the

interface section or in the **implementation** section.

Note: This choice is not relevant for C# and so the selection is not available when refactoring C# code.

5. Click **OK**.

The `uses` or `using` clause is updated with the selected units or namespaces.

See Also

Refactoring Overview (see page 57)

2.4.9 Using Bookmarks

You can mark a location in your code with a bookmark and jump directly to it from anywhere in the file. You can set up to ten bookmarks. Bookmarks are preserved when you save the file and available when you reopen the file in the **Code Editor**.

To set a bookmark

1. In the **Code Editor**, right-click the line of code where you want to set a bookmark. The **Code Editor** context menu is displayed.
2. Choose **Toggle Bookmarks**▶**Bookmark n**, where *n* is a number from 0 to 9. A bookmark icon is displayed in the left gutter of the **Code Editor**.

   Tip: To set a bookmark using the shortcut keys, press CTRL+SHIFT

   and a number from 0 to 9.

To jump to a bookmark

1. In the **Code Editor**, right-click to display the context menu.
2. Choose **GoTo Bookmarks**▶**Bookmark n**, where *n* is a number from 0 to 9.

   Tip: To jump to a bookmark using the shortcut keys, press CTRL

   and the number of the bookmark. For example, `CTRL+1` will jump you to the line of code set at bookmark 1.

To remove a bookmark

1. In the **Code Editor**, right-click to display the context menu.
2. Choose **Toggle Bookmarks**▶**Bookmark n**, where *n* is the number of the bookmark you want to remove. The bookmark icon is removed from the left gutter of the **Code Editor**.

   Tip: To remove all bookmarks from a file, choose Clear Bookmarks

   .

2.4.10 Using Class Completion

Class completion automates the definition of new classes by generating skeleton code for Delphi class members that you declare.

To use class completion

1. In the **Code Editor**, declare a class in the interface section of a unit. For example, you might enter the following:

```
type TMyButton = class(TButton)
    property Size: Integer;
    procedure DoSomething;
end;
```

2. Right-click on the class declaration and choose Complete Class at Cursor.

   Tip: You can also invoke Class Completion by placing the cursor within the class declaration and pressing CTRL+SHIFT+C

   .

Class Completion automatically adds the **read** and **write** specifiers to the declarations for any properties that require them, and then adds skeleton code in the implementation section for each class method.

Tip: You can also use class completion to fill in interface declarations for methods that you define in the implementation section.

After invoking class completion, the sample code above appears as follows:

```
type TMyButton = class(TButton)
    private
        FSize: Integer;
        procedure SetSize(const Value: Integer);
    published
        property Size: Integer read FSize write set_Size;
        procedure DoSomething;
end;
```

The following skeleton code is added to the implementation section:

```
{ TMyButton }

procedure TMyButton.DoSomething;
begin

end;

procedure TMyButton.SetSize(const Value: Integer);
begin
  FSize := Value;
end;
```

If your declarations and implementations are sorted alphabetically, class completion maintains their sorted order. Otherwise, new routines are placed at the end of the implementation section of the unit and new declarations are placed in private sections at the beginning of the class declaration.

Tip: The Finish Incomplete Properties

option on the Tools ▶ Options ▶ Explorer page determines whether class completion completes property declarations.

2.4.11 Using Code Insight

Code Insight (sometimes referred to as **Code Completion**) is a set of features in the **Code Editor** and the HTML **Tag Editor** that provide code completion, display code parameter lists, and show tool tips for expressions and symbols.

The hint window list box filters out all interface method declarations that are referred to by property read or write clauses. The list box displays only properties and stand-alone methods declared in the interface type.

To enable Code Insight (general task)

1. Choose Tools ▶ Options ▶ Code Insight.
2. On the **Code Insight** page, review and set the Code Insight options and color preferences as needed. Tasks that follow this

one in this help topic give more details about some of the **Code Insight** settings.

3. Click **OK**.

To enable and use Code completion

1. Choose **Tools** ▶ **Options** ▶ **Code Insight**.
2. On the **Code Insight** page, check **Code Completion**.
3. To display a list of types, properties, methods, and events in the **Code Editor**, type either a dot (.) (for Delphi or C++) or an arrow (—>) (for C++) following the name of an object or class name. To display the properties, methods, and events available in a class, type the name of a variable and then press `Ctrl + Space`.
4. Select the displayed element that you want to complete the class or object, and press `ENTER`. To cancel the code completion, either `Backspace` or press `Esc`.

Code Insight Examples

1. If you're using the C++ language, type the name of a variable that represents a pointer to a class instance followed by `Ctrl + Space` to display the properties, methods, and events available in the class. To invoke code completion for a pointer type, the pointer must first be de-referenced. For example, type `this` for C++ or `self`for Delphi.
2. If you're using the C++ language, type an arrow (->) for a pointer to an object. You can also type the name of non-pointer types followed by a period (.) to see its list of inherited and virtual properties, methods, and events. For example, for Delphi, type: `var test: TRect;: : begintest`. For C++, type `TRect test; test`.
3. Type an assignment operator or the beginning of an assignment statement and press `Ctrl + Space` to display a list of possible values for the variable.
4. Type a procedure, function, or method call and press `Ctrl + Space` to display the method and it's list of arguments.
5. Type a record to display a list of fields. (This is the same as Step 1, but uses records instead of classes.)

To enable and use Code parameters

1. Choose **Tools** ▶ **Options** ▶ **Code Insight**.
2. Check the **Code parameters** check box.
3. To use Code completion to display the method arguments in the **Code Editor**, type a method name and an open parenthesis (().

To enable and use ToolTip expression evaluation

1. Choose **Tools** ▶ **Options** ▶ **Code Insight**.
2. Check the **ToolTip expression evaluation** check box.
3. To display the current value of a variable while your program has paused during debugging, point the mouse cursor to any variable name displayed on the **Code Editor**.

To enable and use ToolTip Symbol Insight

1. Choose **Tools** ▶ **Options** ▶ **Code Insight**.
2. Check the **ToolTip symbol insight** check box.
3. While editing your code in the **Code Editor**, point the mouse cursor to any identifier to display its declaration.

See Also

Using Live Templates (see page 148)

Using Code Folding (see page 137)

Customizing Code Editor (see page 141)

Using the HTML Tag Editor

usingcodeinsight.xml

2.4.12 Using Live Templates

Live templates are reusable code statements that are accessible from the **Code Editor**. You can insert pre-defined code segments into your code or add your own code snippets to the **Template** window.

Note: If a template has one or more jump points that are editable, it will automatically enter **SyncEdit** mode when you are inserting it into your code. The jump points allow you to navigate between different areas of the template, using the Tab

key and `SHIFT+Tab` keys. Pressing `ESC`, `Enter`,(or pressing the`Tab` key) from the last jump point exits **SyncEdit** mode and puts the **Code Editor** back into regular edit mode. See the link at the end of this topic for more information about **SyncEdit**.

To insert an existing Live Template into your code:

1. In the **Code Editor**, choose **View ▶ Templates** .
2. Expand the tree in the **Template Manager** for the language you are using, by clicking the plus sign in front of language name.
3. Put the cursor at the place in your code where you want to add the template.
4. Choose the template you want to use in the **Template Manager** window.
5. Click the Execute button in the **Template Manager** window.

After you have inserted a template, you will probably need to fill in data, variables, methods, or other information that is specific to your code. You can use the **Code Completion** feature with some of the templates, as described below.

To use Code Completion with your template:

1. Place your cursor at a jump point in your template.
2. Pres `Ctrl + Space` to invoke the **Code Completion** window.

To surround text with a template using the mouse:

1. Select the code in the **Code Editor** that you want the template to surround.
2. Click the right mouse button and choose the Surround command. This will give you a choice of 'surround-able' templates.
3. Choose a template from the list.

To surround text with a template using the Template Manager window:

1. In the **Code Editor**, choose **View ▶ Templates**.
2. Expand the tree in the **Template Manager** for the language you are using, by clicking the plus sign in front of language name.
3. Choose the template you want to use in the **Template Manager** window.
4. Select the code in the **Code Editor** that you want the template to surround.
5. Click the `Execute` button in the **Template Manager** window.

See Also

Creating Live Templates (↗ see page 138)

Using SyncEdit (↗ see page 150)

Using Code Folding (↗ see page 137)

Customizing the Code Editor (↗ see page 141)

Using Code Insight (↗ see page 146)

Using the HTML Tag Editor

2.4.13 Using the History Manager

The **History Manager** lets you view and compare versions of a file, including multiple backup versions, saved local changes, and the edit buffer of unsaved changes.

For simplicity, the following procedures uses a small text file to introduce the functionality of the **History Manager**. However, the **History Manager** is available for most files, including source code and HTML files.

To create and display file versions in the Contents page

1. Choose **Tools ▶ Options ▶ Editor Options** page and verify that the **Create Backup Files** option is checked.
2. Choose **File ▶ New ▶ Other ▶ Other Files ▶ Text** and click **OK** to display a blank text file in the **Code Editor**.
3. On line one of the file, type `First line of text` and save the file using any name and location.
4. On line two, type `Second line of text` and save the file.
5. On line three, type `Third line of text` and save the file. There are now three versions of the file stored in the current directory in a hidden directory named `__history`.
6. Click the **History** tab, which is next to the **Code** tab. The revision list at the top of the **Contents** tab displays three versions of the file. The first version is named `~1~`, the second is named `~2~`, and the current version is named `File`. The source viewer at the bottom of the tab displays the source for the selected version.
7. Select the different versions to display their source in the source viewer.
8. Click the **Code** tab to return to the **Code Editor** and on line four of the file, type `Fourth line of text` but **do not** save the file. Your change is stored in the editor buffer, but not saved to the file.
9. Review the following toolbar and icon descriptions and then use the next procedure to compare the file versions that you just created.

   Tip: To sort a column on any page of the History Manager

   , click the column heading. The toolbar at the top of the **History Manager** contains the following buttons. Not all buttons are available on all pages of the **History Manager**.

   Tip: The toolbar button functions are also available of the right-click context menus of the History Manager

   pages.

To compare file versions using the Diff page

1. Using the file that you created in the previous procedure, click the **History** tab.
2. Click the **Diff** tab at the bottom of the **History Manager**. The **Differences From** and **To** panes at the top of the page shows the file versions that you can compare. At the bottom of the page, source lines that were deleted are highlighted and marked with a minus sign (–). Lines that were added are highlighted and marked with a plus sign (+). The highlighting colors depend on the **Code Editor** colors.
3. Select the different file versions in both the **Differences From** pane and the **To** pane to see the results in source viewer.

To make a prior file version the current version

1. Using the file from the previous procedures, click the **Contents** tab.
2. Right-click the `~2~` version of the file and select Revert, or click the toolbar button. The **Confirm** dialog box indicates that reverting the file will lose any unsaved changes in the buffer.
3. Click **Yes** on **Confirm** dialog box. The `~2~` version becomes the current version.
4. Return to the **Code Editor** and save the change.

Tip: The Revert command is also available on the Info page.

See Also

IDE Tour (see page 34)

History Manager

2.4.14 Using Sync Edit

The Sync Edit feature lets you simultaneously edit indentical identifiers in selected code. For example, in a procedure that contains three occurrences of `label1`, you can edit just the first occurrence and all the other occurrences will change automatically.

To use Sync Edit

1. In the **Code Editor**, select a block of code that contains identical identifiers.
2. Click the **Sync Edit Mode** icon that appears in the left gutter. The first identical identifier is highlighted and the others are outlined. The cursor is positioned on the first identifier. If the code contains multiple sets of indentical identifiers, you can press `TAB` to move between each identifier in the selection.
3. Begin editing the first identifier. As you change the identifier, the same change is performed automatically on the other identifiers. By default, the identifier is replaced. To change the identifier without replacing it, use the arrow keys before you begin typing.
4. When you have finished changing the identifiers, you can exit Sync Edit mode by clicking the **Sync Edit Mode** icon, or by pressing the`Esc` key.

   Note: Sync Edit determines indentical identifiers by matching text strings; it does not analyze the identifiers. For example, it does not distinguish between two like-named identifiers of different types in different scopes. Therefore, Sync Edit is intended for small sections of code, such as a single method or a page of text. For changing larger sections of code, consider using refactoring.

2.5 Getting Started Procedures

This section provides how-to information on configuring the IDE, working with forms and projects, and more.

Topics

Name	Description
Adding Components to a Form (see page 152)	
Adding References (see page 153)	You can integrate your legacy COM servers and ActiveX controls into managed applications by adding references to unmanaged DLLs to your project, and then browse the types just as you would with managed assemblies.
Adding and Removing Files (see page 153)	You can add and remove a variety of file types to your projects.
Adding Templates to the Object Repository (see page 153)	You can add your own objects to the **Object Repository** as templates to reuse or share with other developers. Reusing objects lets you build families of applications with common user interfaces and functionality to reduce development time and improve quality.
Copying References to a Local Path (see page 154)	During runtime, assemblies must be in the output path of the project or in the GAC for deployment. If your project contains a reference to an object that is not in one of the two locations, the reference must be copied to the appropriate output path.
Creating a Component Template (see page 154)	You can save selected, preconfigured components on the current form as a reusable component template accessible from the **Tool Palette**.
Creating a Project (see page 155)	
Customizing the Form (see page 155)	
Customizing the Tool Palette (see page 156)	
Customizing Toolbars (see page 156)	
Disabling Themes in the IDE and in Your Application (see page 157)	Both Windows Vista and Windows XP support themes in the user interface. By default, the IDE uses themes, and runtime themes are enabled for the application itself. If you prefer or if you require the classic user interface style, you can disable the use of themes in the IDE and in your application.
Docking Tool Windows (see page 157)	The Auto-Hide feature lets you undock and hide tool windows, such as the **Object Inspector**, **Tool Palette**, and **Project Manager**, but still have access to them.
Finding Items on the Tool Palette (see page 158)	
Exploring .NET Assembly Metadata Using the Reflection Viewer (see page 158)	You can open and explore the namespaces and types contained with a .NET assembly. The assembly metadata is displayed in the Reflection viewer, whose left pane contains a tree structure of the namespaces and types within the assembly. The right pane displays specific information on the selected item in the tree. The **Call Graph** tab shows you a list of the methods called by the selected method, as well as a list of the methods that call the selected method. You can open multiple .NET assemblies in the Reflection viewer. Each open assembly is displayed in the tree in the... more (see page 158)
Exploring Windows Type Libraries (see page 159)	You can open and inspect the interfaces and other types contained within a Windows type library. The type library contents are displayed in a Windows Explorer-style presentation, with a left pane containing a tree of the interface and type definitions within the type library. The right pane displays specific information on the selected item in the tree. The **Type Library Explorer** can open a `.TLB` file, as well as OCX controls, and `.DLL` and `.EXE` files that have type libraries as embedded resources.
Installing Custom Components (see page 160)	
Renaming Files Using the Project Manager (see page 160)	Renaming a file changes the name of the file in both the **Project Manager** and on disk.
Saving Desktop Layouts (see page 161)	To switch between desktop layouts, choose a layout from the drop-down list box located on the **Desktop** toolbar. This procedure describes how to save your current desktop layout so that your own layout is available from the **View** ▶ **Desktops** submenu and from the toolbar.
Setting Component Properties (see page 161)	After you place your components on your Designer, set their properties using the **Object Inspector**. By setting a component's properties, you can change the way a component appears and behaves in your application. Because properties appear during designtime, you have more control over a component's properties and can easily modify them without having to write additional code.

Setting Dynamic Properties (see page 161)	Many of the .NET Framework objects support dynamic properties. Dynamic properties provide a way to change property values without recompiling an application. The dynamic properties and their values are stored in a configuration file, along with the application's executable file. Changing a property value in the configuration file causes the change to take effect the next time the applications runs. Dynamic properties are useful for changing an application after it has been deployed.
Setting Project Options (see page 162)	You can manage application and compiler options for your project. Making changes to your project only affects the current project. However, you can also save your selections as the default settings for new projects.
Setting C++ Project Options (see page 163)	You can manage application and compiler options for your project. Making changes to your project only affects the current project. However, you can also save your selections as the default settings for new projects.
Setting Properties and Events (see page 164)	Properties, methods, and events are attributes of a component.
Setting The IDE To Mimic Delphi 7 (see page 164)	Use this procedure to set the IDE to mimic Delphi 7 or C++Builder, where each pane is its own window.
Setting Tool Preferences (see page 165)	You can customize the appearance and behavior of many tools and features, such as the **Object Inspector**, **Code Editor**, and integrated debugger.
Using Design Guidelines with VCL Components (see page 165)	You can use VCL or VCL.NET (with Delphi or C++) to setup components that are "aware" of their relation to other components on a form. You can set properties to specify the distance between controls, shortcuts, focus labels, tab order, and maximum number of items (listboxes, menus).
Using the File Browser (see page 166)	The File Browser is a standard Windows-style browser that you can undock within the IDE. The context menu on the File Browser enables you to perform file operations such as Cut, Copy, Delete and Rename. You can also add a file to your project using the Add to project command.
Using To-Do Lists (see page 166)	A to-do list records and displays tasks that need to be completed for a project.
Using Virtual Folders (see page 167)	For C++ only, the IDE provides the ability to organize your project files with virtual folders in the **Project Manager** tree. Virtual folders only affect the display of the folder structure in the IDE. Moving files into virtual folders does not change their actual location on disc. **Note:** Virtual folders can only contain file system entries or other virtual folders. **Note:** Changing the order of entries in a virtual folder changes the build order of the contained buildable entries.
Writing Event Handlers (see page 168)	Your source code usually responds to events that might occur to a component at runtime, such as a user clicking a button or choosing a menu command. The code that responds to an occurrence is called an event handler. The event handler code can modify property values and call methods.

2.5.1 Adding Components to a Form

To add components to a form

1. On the **Tool Palette**, select a visual or nonvisual component.
2. Double-click the component to place it on the form or drag the component onto the form. If you add a nonvisual component to the form, the component tray appears at the bottom of the Designer surface.
3. Repeat steps 1 and 2 to add additional components.
4. Use the dotted grid on the form to align your components.

See Also

Getting Started

Starting a Project (see page 47)

Creating a Project (see page 155)

Setting Project Options (see page 162)

Setting Properties and Events (see page 164)

2.5.2 Adding References

You can integrate your legacy COM servers and ActiveX controls into managed applications by adding references to unmanaged DLLs to your project, and then browse the types just as you would with managed assemblies.

To add references

1. From the main menu, choose **Project** ▶ **Add Reference**. The **Add Reference** dialog box appears.
2. Select either a legacy COM type library or ActiveX control to integrate into your managed application.
3. Click **Add Reference**. The reference is added to the text box.
4. Click **OK**.

   Tip: You can also right-click the References

   folder in the **Project Manager**, and choose Add Reference.

2.5.3 Adding and Removing Files

You can add and remove a variety of file types to your projects.

To add a file to a project

1. Choose **Project** ▶ **Add to Project**. The **Add to Project** dialog box appears.
2. Select a file to add and click **Open**. The file appears below the `Project.exe` node of the **Project Manager**.

To remove a file from a project

1. Choose **Project** ▶ **Remove From Project**. A **Remove From Project** dialog box appears.
2. Select the file or files you want to remove and click **OK**.

See Also

Getting Started

Creating a Project (↗ see page 155)

2

2.5.4 Adding Templates to the Object Repository

You can add your own objects to the **Object Repository** as templates to reuse or share with other developers. Reusing objects lets you build families of applications with common user interfaces and functionality to reduce development time and improve quality.

To add a template to the Object Repository

1. Save your project.
2. Choose **Project** ▶ **Add to Repository**.
3. Enter the project name, description, and author information in the dialog box.

4. Click **Browse** to select an icon to represent the project you saved.
5. Click **OK**.

See Also

Getting Started

Adding and Removing Files (see page 153)

2.5.5 Copying References to a Local Path

During runtime, assemblies must be in the output path of the project or in the GAC for deployment. If your project contains a reference to an object that is not in one of the two locations, the reference must be copied to the appropriate output path.

To a copy reference to a local path

1. In the **Project Manager**, right-click an assembly DLL in the **References** folder.
2. Set the **Copy Local** option to copy the file to the output directory.

   Note: The IDE maintains the Copy Local

   setting until you change it.

See Also

Tour of the IDE (see page 34)

2.5.6 Creating a Component Template

You can save selected, preconfigured components on the current form as a reusable component template accessible from the **Tool Palette**.

To create a component template

1. Place and arrange components on a form.
2. In the **Object Inspector**, set the component properties and events as desired.
3. Select the components that you want to save as a component template. To select several components, drag the mouse over them.

   Tip: To select all of the components on the form, choose Edit->Select All

   . Gray handles appear at the corners of each selected component.
4. Choose Component ▶ Create Component Template. The **Create Component Template** dialog box appears.
5. Specify a name, a **Tool Palette** category, and an icon for the template.
6. Click **OK**.

Your new template appears immediately on the **Tool Palette**, in the category that you specified.

To use a component template

1. Display the form to which you want to add the components from the component template.
2. On the **Tool Palette**, double-click the component template icon. The components in the component template are added to the form, along with their preconfigured properties and events. You can reposition the components independently, reset their

properties, and create or modify event handlers for them, just as if you had placed each component in a separate operation.

To delete a component template

1. On the **Tool Palette**, right-click the component template to display a context menu.
2. Choose the Delete [template name] Button command. The component template is deleted immediately from the **Tool Palette**.

2.5.7 Creating a Project

To add a new project

1. Choose **Project▶Add New Project**. The **New Items** dialog box appears.
2. Select a project and click **OK**. The project is added to the **Project Manager**.

To add an existing project

1. Choose **Project▶Add Existing Project**. The **Open Project** dialog box appears.
2. Select an existing project to add and click **Open**.

See Also

Starting a Project (see page 47)

Adding and Removing Files (see page 153)

Adding Components (see page 152)

Setting Project Options (see page 162)

Setting Properties (see page 164)

2.5.8 Customizing the Form

To customize the form

1. Choose **Tools▶Options**.
2. From the **Options** dialog box, click **Windows Forms Designer**.
3. Enable or disable the snap to grid and show grid features by selecting and deselecting the check boxes.
4. Choose one of the bracing styles.
5. Click **OK**.

   Tip: The changes will affect only forms created after these options are changed. To change the settings for existing forms, set the GridSize, DrawGrid, and SnapToGrid properties of the form.

See Also

Tour of the IDE (see page 34)

Starting a Project (see page 47)

Adding Components (see page 152),

2.5.9 Customizing the Tool Palette

To arrange individual components

1. Click the component.
2. Drag the component anywhere within the **Tool Palette**.

To arrange an entire category of components

1. Click a category name .
2. Drag the category anywhere within the **Tool Palette**.
3. Release your mouse button to place the category in the desired location.

To add additional categories

1. Right-click the **Tool Palette**.
2. Choose the **Add New Category** command. The **Create a new Category** dialog box appears.
3. Enter a name for the category in the **New Category Name** text box.
4. Click **OK**. The new category appears at the bottom of the **Tool Palette**.

See Also

Creating a Component Template (↗ see page 154)

2.5.10 Customizing Toolbars

To arrange your toolbars

1. Click the grab bar on the left side of any toolbar.
2. Drag the toolbar to another location or onto your desktop.

To delete icons from the toolbar

1. Choose **View ▶ Toolbars ▶ Customize**.
2. From the toolbar, not the **Customize** dialog box, drag the tool from the toolbar until its icon displays an X and then release the mouse button.
3. When completed, click **Close**.

To add icons to the toolbar

1. Choose **View ▶ Toolbars ▶ Customize**.
2. Click the **Commands** tab.
3. In the **Categories** list, select a category to view its tool icons.
4. From the **Commands** list, drag the selected icon onto the toolbar of your choice.
5. When completed, click **Close**.

See Also

Tour of the IDE (↗ see page 34)

Customizing the Tool Palette (see page 156)

2.5.11 Disabling Themes in the IDE and in Your Application

Both Windows Vista and Windows XP support themes in the user interface. By default, the IDE uses themes, and runtime themes are enabled for the application itself. If you prefer or if you require the classic user interface style, you can disable the use of themes in the IDE and in your application.

To disable themes for the IDE

1. Close the IDE.
2. Open the file `bds.exe.manifest` located in your `$(IDE)\bin` directory.
3. Remove or comment out the following entry in the XML file `bds.exe.manifest`:

```
  <dependency>
   <dependentAssembly>
     <assemblyIdentity
       type="win32"
       name="Microsoft.Windows.Common-Controls"
       version="6.0.0.0"
       publicKeyToken="6595b64144ccf1df"
       language="*"
       processorArchitecture="*"   />
   </dependentAssembly>
</dependency>
```

4. Save the changes to the `bds.exe.manifest` file.
5. Restart the IDE.

To disable theming for an application

1. Choose **Project**▶**Options**▶**Application**.
2. Uncheck **Enable runtime themes**.

See Also

Starting a Project (see page 34)

2.5.12 Docking Tool Windows

The Auto-Hide feature lets you undock and hide tool windows, such as the **Object Inspector**, **Tool Palette**, and **Project Manager**, but still have access to them.

To use Auto-Hide to hide your tools

1. Click the push pin in the upper right corner of a tool window. The tool window is replaced by one or more tabs at the outer edge of the IDE window.
2. To display the tool window, position the cursor over the tab. The tool window slides into view.
3. To slide the tool window out of view, move the cursor away from the tool window.
4. To redock the tool window, click the push pin until it points down.

To dock the tools with one another

1. Click the tool window title bar and drag the window into another tool window.
2. Select a location to drop the tool window and release the mouse button.

To undock the tools from one another

1. Click the tool window title bar and drag the window away from the other tool window.
2. Select a location to drop the tool window and release the mouse button.

See Also

Saving Desktop Layouts (see page 161)

Setting Tool Preferences (see page 165)

2.5.13 Finding Items on the Tool Palette

To find items on the Tool Palette

1. Click anywhere on the **Tool Palette** and start typing the name of the item that you want to find. The **Tool Palette** is filtered to display only those item names that match what you are typing. The characters that you have typed appear in **bold** in the item names.
2. Double-click an item to perform the default action for that item. For example, double-clicking a component adds it to your form, whereas double-clicking a code snippet adds it to your code.
3. To remove the search filter from the **Tool Palette**, click the filter icon .

See Also

Adding Components to the Tool Palette (see page 160)

2.5.14 Exploring .NET Assembly Metadata Using the Reflection Viewer

You can open and explore the namespaces and types contained with a .NET assembly. The assembly metadata is displayed in the Reflection viewer, whose left pane contains a tree structure of the namespaces and types within the assembly. The right pane displays specific information on the selected item in the tree. The **Call Graph** tab shows you a list of the methods called by the selected method, as well as a list of the methods that call the selected method.

You can open multiple .NET assemblies in the Reflection viewer. Each open assembly is displayed in the tree in the left-pane; the top-level node for a .NET assembly is denoted by the clr icon.

There are several ways to open the Reflection viewer:

- Choose File ▸ Open and selecting any managed assembly.
- Use the **Open** context-menu command in the **Project Manager.**
- Use the context-menu **Browse Class** command in the debugger **Modules** window.
- Use the standalone application (Reflection.exe).

To close a particular .NET assembly, right-click the top-level clr icon and select Close.

To open the Reflection viewer from the menu

1. Choose **File**▶**Open**.
2. In the **Open** dialog box, from the **Files of type** drop-down list, select **Assembly Metadata**.
3. Navigate to the folder where a .NET assembly is located. Select the assembly and click **Open**.

   Tip: You can use the Browser buttons on the toolbar to navigate backwards and forwards to previously selected items in the left pane.

To open the Reflection viewer from the Project Manager

1. Open a .NET application, such as an application for Delphi for .NET, for ASP.NET Web, or for VCL for .NET.
2. In the **Project Manager**, right-click an assembly such as `System.Data.dll`.
3. Select Open from the context menu.

The **Reflection** viewer, displayed in the **Code Editor** pane, contains tabs that correspond to the type of the item that is currently selected in the lefthand column of the viewer. When opened from the **Project Manager**, the **Reflection** viewer itself has a tab labeled with the class name, such as **System.Data.dll**.

To open the Reflection viewer from the debugger

1. Open a .NET application, such as an application for Delphi for .NET, ASP.NET, or VCL for .NET.
2. Press F8 to start the debugger.
3. Select **View**▶**Debug Windows**▶**Modules**.
4. Select any item in the Modules view. This populates the scope browser.
5. In the scope browser, right-click a class, represented by the class icon: ●
6. Select the **Browse Class** command from the context menu.

The **Reflection viewer**, displayed in the **Code Editor** pane, contains tabs that correspond to the type of the item that is currently selected in the **Modules** window. When opened from the **Modules** window, the **Reflection viewer** itself has a tab labeled **Browse<itemname>**.

Using the Call Graph tab

1. Select a method node in the left pane.
2. Select the **Call Graph** tab. The top half of the **Call Graph** tab shows you a list of methods that call the method you selected in the left pane. The bottom half of the **Call Graph** tab shows you the methods called by the method you selected in the left pane. Methods that exist in the same assembly as the currently selected method appear as clickable links, and are displayed in blue underlined text. Clicking on a link causes that method to become selected in the tree in the left-hand pane.

   Note: The standalone Reflection

   viewer (Reflection.exe) has a **Find** button for searching an assembly, and two arrow buttons that move the viewer forward and backward in the viewer history.

See Also

Assembly Metadata Explorer (IDE Reference) (↗ see page 1063)

2.5.15 Exploring Windows Type Libraries

You can open and inspect the interfaces and other types contained within a Windows type library. The type library contents are displayed in a Windows Explorer-style presentation, with a left pane containing a tree of the interface and type definitions within the type library. The right pane displays specific information on the selected item in the tree. The **Type Library Explorer** can open a `.TLB` file, as well as OCX controls, and `.DLL` and `.EXE` files that have type libraries as embedded resources.

To Inspect a Windows Type Library

1. Choose **File ▶ Open**.
2. In the **Open** dialog box, from the **Files of type** drop-down list, select **Type Library**. This sets the file filter to display files with extensions of `.TLB`, `.OLB`, `.OCX`, `.DLL`, and `.EXE`.
3. Navigate to the folder where the type library is located.
4. Select the file and click **Open**.

You can open multiple type libraries in the explorer. Each open type library is displayed in the tree in the left pane; the top-level node for a type library is denoted by the ❖ icon.

To close a particular type library, right-click on the top-level ❖ icon and select Close.

See Also

Type Library Explorer (IDE Reference) (see page 1065)

2.5.16 Installing Custom Components

To install custom components

1. Choose **Component ▶ Installed .NET Components**.
2. Click **Select an Assembly**.
3. Navigate to the folder containing the component assembly. Alternatively, you can enter the name of the full path to the assembly in the **File Name** field.
4. Select the assembly.
5. Click **Open**. The **Installed .NET Components** dialog box displays the components from the assembly.
6. Verify that the components you want to install on the **Tool Palette** are checked.
7. Click **OK**.

See Also

Adding Components (see page 152)

2.5.17 Renaming Files Using the Project Manager

Renaming a file changes the name of the file in both the **Project Manager** and on disk.

To rename a file

1. In the **Project Manager**, right-click the file that you want to rename. The context menu is displayed.
2. Choose **Rename**.
3. Enter the new name for the file. If the file has associated files that appear as child nodes in the **Project Manager** tree, those files are automatically renamed.

See Also

Tour of the IDE (see page 34)

2.5.18 Saving Desktop Layouts

To switch between desktop layouts, choose a layout from the drop-down list box located on the **Desktop** toolbar.

This procedure describes how to save your current desktop layout so that your own layout is available from the **View**▶**Desktops** submenu and from the toolbar.

To save a desktop layout

1. Choose **View**▶**Desktops**▶**Save Desktop**.
2. Enter the name you want for the desktop.
3. Click **OK**.

To set a Debug desktop layout

1. Choose **View**▶**Desktops**▶**Set Debug Desktop**.
2. From the dropdown list, select the layout you want to use as your Debug desktop layout.
3. Click **OK**.

See Also

Setting Project Options (see page 162)

Overview of Debugging (see page 10)

Desktop Toolbar (see page 1036)

View Desktops Command (see page 1059)

2.5.19 Setting Component Properties

After you place your components on your Designer, set their properties using the **Object Inspector**. By setting a component's properties, you can change the way a component appears and behaves in your application. Because properties appear during designtime, you have more control over a component's properties and can easily modify them without having to write additional code.

To set component properties

1. On the **Object Inspector**, click the **Properties** tab.
2. Set the component properties by entering values in the text box or through an editor. Boolean properties like **True** and **False** can be toggled.

See Also

Creating a Project (see page 155)

2.5.20 Setting Dynamic Properties

Many of the .NET Framework objects support dynamic properties. Dynamic properties provide a way to change property values without recompiling an application. The dynamic properties and their values are stored in a configuration file, along with the application's executable file. Changing a property value in the configuration file causes the change to take effect the next time the applications runs. Dynamic properties are useful for changing an application after it has been deployed.

To set a dynamic property in the Object Inspector

1. In a form on the **Design** tab, click the object for which you want to set dynamic properties.
2. In the **Object Inspector**, expand **(DynamicProperties)** and click **(Advanced)**. If the object does not support dynamic properties, **(DynamicProperties)** is not displayed.

   Tip: If the Object Inspector

   is arranged by category, **(DynamicProperties)** is displayed under **Configurations**.
3. Click the **ellipsis (...)** button next to **(Advanced)** to display the **Dynamic Properties** dialog box. This dialog lists all of the properties that can be stored in the configuration file.
4. Select the properties you want to store in the configuration file.
5. Optionally, you can override the default key name listed in the **Key mapping** field.
6. Click **OK**. The dynamic properties are marked with an icon in the **Object Inspector**. RAD Studio creates an XML file named `app.config` (for a Windows application) or `Web.config` (for a Web application) in the project directory. This file lists the dynamic properties and their current values.
7. Compile the application. RAD Studio creates a file named `<projectname>.exe.config` (for a Windows application) or `<projectname>.dll.config` (for a Web application) in the same directory as the application's executable or DLL file.

To change a dynamic property value in the configuration file

1. In the directory that contains the application's executable or DLL file, locate the configuration file.
2. Open the file in a text editor.
3. Locate the `add key=` statement for the property to be changed and edit the value.
4. Save your changes and close the file.

The next time the application runs, the changed property value will be in effect.

See Also

Introduction to Dynamic Properties

2.5.21 Setting Project Options

You can manage application and compiler options for your project. Making changes to your project only affects the current project. However, you can also save your selections as the default settings for new projects.

To change compiler options

1. Choose **Project**▶**Options**. The **Options** dialog box appears.
2. Select **Compiler** and set your options to modify how you want your program to compile.
3. Click **OK**.

To change application options

1. Choose **Project**▶**Options**. The **Options** dialog box appears.
2. Select **Application** and specify a title and extension for your application.
3. Click **OK**.

To change debugger options

1. Choose **Project▶Options**. The **Options** dialog box appears.
2. Use the **Debugger** page to pass command-line parameters to your application, specify a host executable for testing a DLL, or load an executable into the debugger.
3. Use the **Environment Block** page to indicate which environment variables are passed to your application while you are debugging it.
4. Click **OK**.

See Also

Tour of the IDE (see page 34)

Adding Components (see page 152)

Adding and Removing Files (see page 153)

Creating a Project (see page 155)

Setting Properties (see page 164)

Build Configurations Overview (Delphi) (see page 5)

Build Configurations Overview (C++) (see page 6)

Named Option Sets Overview (see page 7)

2.5.22 Setting C++ Project Options

You can manage application and compiler options for your project. Making changes to your project only affects the current project. However, you can also save your selections as the default settings for new projects.

To change option values

1. Choose **Project▶Options**. The **Options** dialog box appears.
2. Select a page from the list in the left pane.
3. If you leave the cursor over text describing an option, a tool tip gives you an option description, its default value, and a switch for the option if one exists.
4. If an option's value differs from its parent configuration's value, its associated text is boldface.
5. Set options on the page to determine how your project is built. Depending on the option, you may enter text, check or uncheck a box, or make a selection from a pull down menu. Some options have an ellipsis box that you click to display a dialog to choose a file or directory or a dialog to manage a list of items, such as paths.
6. Some options that contain a list of items, such as defines or paths, have a **Merge** check box. If checked, the IDE merges the option's list with that of its immediate ancestor's configuration's list for that option. Note that the IDE does not actually change the contents of the option, but acts as if the list included the ancestor's list. If the ancestor's **Merge** check box is also checked, the IDE also merges this ancestor's list for that option, and so on up the inheritance chain. If unchecked, the IDE uses only the items in the current configuration.
7. Click **OK** to accept the changes and close the dialog. Click **Cancel** to ignore the changes and close the dialog.

To revert option values to the parent configuration's value

1. Choose **Project▶Options**. The **Options** dialog box appears.
2. Select a page from the list in the left pane and set your options to determine how you want your project to be built.

3. If an option's value differs from its parent configuration's value, its associated text is boldface.
4. Right-click the option and click **Revert** on the context menu. The option value changes to the parent configuration's value.
5. Click **OK** to accept the changes and close the dialog. Click **Cancel** to ignore the changes and close the dialog.

See Also

Tour of the IDE (see page 34)

Adding Components (see page 152)

Adding and Removing Files (see page 153)

Creating a Project (see page 155)

Setting Properties (see page 164)

2.5.23 Setting Properties and Events

Properties, methods, and events are attributes of a component.

To set object properties

1. On your form, click once on the object to select it.
2. In the **Object Inspector**, click the **Properties** tab.
3. Select the property that you want to change and either enter a value in the text box, select a value from the drop-down list, or click the ellipsis (...) next to the text box to use the associated property editor, depending on which update technique is available for the property.

To set an event handler

1. On your form, click once on the object to select it.
2. On the **Object Inspector**, click the **Events** tab.
3. If an event handler already exists, select it from the drop-down box. Otherwise, double-click the event to switch to **Code** view.
4. Type the code you want to execute when the event occurs.

See Also

Tour of the IDE (see page 34)

Adding Components (see page 152)

Adding and Removing Files (see page 153)

Creating a Project (see page 155)

Setting Project Options (see page 162)

2.5.24 Setting The IDE To Mimic Delphi 7

Use this procedure to set the IDE to mimic Delphi 7 or C++Builder, where each pane is its own window.

To turn off the Embedded Designer layout

1. Choose **Tools**▶**Options**▶**Environment Options**▶**VCL Designer**.

2. Uncheck **Embedded Designer**.
3. Click **OK**.
4. Restart RAD Studio for the change to take effect.

2.5.25 Setting Tool Preferences

You can customize the appearance and behavior of many tools and features, such as the **Object Inspector**, **Code Editor**, and integrated debugger.

To set tool preferences

1. Choose **Tools ▶ Options**.
2. Review the options in each tool category and customize the settings to suit your needs.
3. Click **OK**.

See Also

Customizing the Form (see page 155)

Customizing the Tool Palette (see page 156)

Setting Project Options (see page 162)

2.5.26 Using Design Guidelines with VCL Components

You can use VCL or VCL.NET (with Delphi or C++) to setup components that are "aware" of their relation to other components on a form. You can set properties to specify the distance between controls, shortcuts, focus labels, tab order, and maximum number of items (listboxes, menus).

To see and use the design guidelines:

1. Register an object type.
2. Indicate various points on or near a component's bounds that are "alignment" points. These "alignment" points are vertical or horizontal lines that cut across a visual control's bounds.
3. Supply UI guideline information so that each component will adhere to rules such as distance between controls, shortcuts, focus labels, tab order, maximum number of items (listboxes, menus),

Your new Error Reconcile Form will display four columns in the upper portion of the window, and six radio buttons in the bottom portion of the window. The following table describes each of the columns.

Component	Default Value when 'Use Design Guidelines' is Set
Alignment	The names of the columns of the table in which an error has occurred.
Margins	Bottom = 3, Left = 3, Right = 3, Right = 3, Top = 3
Padding	The last update that was saved to the Server. (This represents what the row contains on the server.)

See Also

UI Design (see page 15)

Tour of the IDE (see page 34)

2.5.27 Using the File Browser

The File Browser is a standard Windows-style browser that you can undock within the IDE. The context menu on the File Browser enables you to perform file operations such as Cut, Copy, Delete and Rename. You can also add a file to your project using the Add to project command.

To open and use the File Browser

1. In the IDE, choose **View** ▶ **File Browser**.
2. Right-click a file name and select the appropriate command from the context menu. You can open files within RAD Studio, add selected files to the current project, or perform standard Windows operations on files.

To filter the file list

1. Click the **Set Filter** icon in the menu bar of the File Browser.
2. In the **Set Filter**dialog box, enter file names or wild-card expressions, separated by semicolons. The File Browser displays any files that match the wild-card expansion of any of the filter criteria you entered.

See Also

File Browser (↗ see page 1036)

2.5.28 Using To-Do Lists

A to-do list records and displays tasks that need to be completed for a project.

To create a to-do list and add an item to it

1. Choose **View** ▶ **To-Do List**.
2. In the **To-Do List** dialog box, right-click and choose Add.
3. In the **Add To-Do Item** dialog box, enter a description of the task and adjust the other fields as necessary.
4. Click **OK**.

To add a to-do list item as a comment in code

1. In the **Code Editor**, position your cursor where you want to add the comment.
2. Right-click and choose **Add To-Do List Item**.
3. In the **Add To-Do Item** dialog box, select the item that you want to add.
4. Click **OK**.

The item is added as a comment to your code, beginning with the word `TODO`.

To mark a to-do list item as completed

1. Choose **View** ▶ **To-Do List**.
2. In the **To-Do List** dialog box, check the check box next to the item to indicate completion. The item remains in the list, but the text is crossed out. If the item was added as a comment to code, the comment is updated to indicate `DONE` instead of `TODO`.

To filter the items in a to-do list

1. Choose View ▶ To-Do List.
2. Right-click the **To-Do List** dialog box and choose Filter.
3. Choose either Categories, Owner, or Item types, depending on which you want to filter.
4. In the **Filter To-Do List** dialog box, uncheck the items that you want to hide in the to-do list.
5. Click **OK**. The to-do list is redisplayed, with the filtered items hidden. The status bar at the bottom of the **To-Do List** dialog box indicates how many items are hidden due to filtering.

To delete an item from a to-do list

1. Choose View ▶ To-Do List.
2. In the **To-Do List** dialog box, select the item to delete.
3. Right-click and choose Delete. The item is removed from the **to-do list**. If the item was added as a comment to code, the comment is also removed.

2.5.29 Using Virtual Folders

For C++ only, the IDE provides the ability to organize your project files with virtual folders in the **Project Manager** tree. Virtual folders only affect the display of the folder structure in the IDE. Moving files into virtual folders does not change their actual location on disc.

Note: Virtual folders can only contain file system entries or other virtual folders.

Note: Changing the order of entries in a virtual folder changes the build order of the contained buildable entries.

To add a root-level virtual folder

1. Choose View ▶ Project Manager to display the **Project Manager** if it is not already visible in the IDE.
2. Create a new project or open an existing one.
3. Right-click the project node in the **Project Manager** tree and choose Add New ▶ Virtual Folder. This opens the **Add new folder** dialog box and displays the default folder name **Virtual folder 1**
4. Type a new name for the virtual folder if you do not want to use the default.
5. Click **OK** to add the folder. Click **Cancel** to not add the folder. The virtual folder appears as a greyed folder under the project node.
6. Drag files from the project structure into the virtual folder, or use the context menu commands to add items to the virtual folder.

To add a virtual sub-folder to an existing virtual folder

1. Right-click an existing virtual folder in the **Project Manager** tree.
2. Choose Add New ▶ Virtual Folder.

To change the order of files in a virtual folder

1. Click or right-click on a file in a virtual folder and drag the file. The cursor changes to an arrow with a rectangle at the end to indicate you are moving the file.
2. Drag the file to the right until a horizontal blue line appears. This line shows the new location of the file in the virtual folder and only appears between files. If the cursor changes to a slashed circle for some location, you can't move the file there.

3. Drag the file to the desired location in the virtual folder and release the mouse button.

To delete a virtual folder

1. Select a virtual folder in the **Project Manager** tree.
2. Right-click and choose Delete.
3. Click **Yes** in the **Confirm** dialog box to delete the folder. Click **No** or press the Esc key to cancel the delete.

   Note: Deleting a virtual folder does not delete any of its files on the disk. It simply removes the folder from the Project Manager

   tree and leaves the files in their original locations.

See Also

Overview of Virtual Folders (see page 50)

Adding and Removing Files (see page 153)

2.5.30 Writing Event Handlers

Your source code usually responds to events that might occur to a component at runtime, such as a user clicking a button or choosing a menu command. The code that responds to an occurrence is called an event handler. The event handler code can modify property values and call methods.

To write an event handler

1. On your form, click the component for which you want to write an event handler.
2. To create the default event for the component, double-click the component on the form. To choose another event for the component, click the **Events** tab in the **Object Inspector**, locate the event, and double-click its text box. The **Code Editor** appears.
3. Type the code that will execute when the event occurs at runtime.

See Also

Creating a Project (see page 155)

Setting Component Properties (see page 161)

2.6 Localization Procedures

This section provides how-to information on localizing applications by using the RAD Studio translation tools.

Topics

Name	Description
Adding Languages to a Project (↗ see page 169)	You can add languages to your project by using the **Satellite Assembly Wizard** (.NET) or **Resource DLL Wizard** (Win32). For each language that you add, the wizard generates a resource module project in your project group. Each resource module project is given an extension based on the language's locale.
Editing Resource Files in the Translation Manager (↗ see page 170)	After you have added languages to your project by using the **Satellite Assembly Wizard** or **Resource DLL Wizard**, you can use the Translation Manager to view and edit your resource files. You can edit resource strings directly, add translated strings to the Translation Repository, or get strings from the Translation Repository.
Setting the Active Language for a Project (↗ see page 171)	After adding languages to your project with the **Satellite Assembly Wizard** or the **Resource DLL Wizard**, the base language module is loaded when you choose **Run** ▶ **Run**. However, you can load a different language module by setting the active language for the project.
Setting Up the External Translation Manager (↗ see page 172)	If you do not have the RAD Studio IDE, you can use the External Translation Manager (ETM) to localize an application. To use ETM, the developer must provide you with the required ETM files and project files. **Note:** The Microsoft .NET Framework must be installed on your computer before you install ETM.
Updating Resource Modules (↗ see page 173)	When you add an additional resource, such as a button on a form, you must update your resource modules to reflect your changes.
Using the External Translation Manager (↗ see page 173)	Translators who do not have the RAD Studio IDE can use the External Translation Manager (ETM) instead of the Translation Manager. The steps for using the ETM are similar to those for the internal Translation Manager. **Note:** ETM must be set up and operational on your computer before using the following procedure. See in the link listed at the end of this topic for details.

2.6.1 Adding Languages to a Project

You can add languages to your project by using the **Satellite Assembly Wizard** (.NET) or **Resource DLL Wizard** (Win32). For each language that you add, the wizard generates a resource module project in your project group. Each resource module project is given an extension based on the language's locale.

To add a language to a project

1. Save and build your project.
2. With your project open in the IDE, choose **Project** ▶ **Languages** ▶ **Add**. Alternatively, you can choose either **File** ▶ **New** ▶ **Other** ▶ **Delphi for .NET Projects** ▶ **Satellite Assembly Wizard** for a .NET application or **File** ▶ **New** ▶ **Other** ▶ **Delphi Projects** ▶ **Resource DLL Wizard** for a Win32 application. The wizard is displayed.
3. Make sure your project is selected in the list that appears in the dialog and then click **Next**.
4. Click the check box next to the languages that you want to add to your project and then click **Next**.
5. Review the directory path information that the wizard will use for the language's resource modules.

   Tip: To change the path, click the path, and then click the ellipsis (...) button to browse to a different directory.

   When you are satisfied with the path information, click **Next.**
6. If no satellite assembly for the language exists yet, **Create New** appears in the **Update Mode** column. Click **Next**. If a resource module exists for the language in the directory you have specified, click in the **Update Mode** column to select

Update or **Overwrite**. Choose **Update** to keep and modify the existing satellite assembly project. Choose **Overwrite** to create a new, empty project and to delete the old project and any translations it contains. Click **Next**.

7. Review the summary of what the wizard will do and click **Finish** to create or update the resource modules for the languages you have selected. If the wizard asks to generate a `.drcil` (.NET) or `.drc` (Win32) file, click **Yes**. Any project that uses its own resource strings (instead of previously compiled `.rc` files) needs a `.drcil` or `.drc` file. If you are sure that no new files are needed (because your project does not introduce any resource strings of its own), select **Skip drcil files that are not found** in the final dialog. This prevents the wizard from generating, or asking to generate, files.
8. Click **Yes** to compile. Click **OK** to save your project group.

The generated projects contain untranslated copies of the resource strings in your original project. By default, the Translation Manager is displayed, enabling you to begin translating the resource files.

To remove a language from a project

1. Open your project.
2. Select **Project ▶ Languages ▶ Remove**.
3. Check the languages that you want to remove and then click **Next**.
4. Click **Finish**.

The wizard removes the selected resource module from your project file, but does not delete the assemblies, the source of the assemblies, or the directories in which they reside.

To restore a language to a project

1. Choose **Project ▶ Languages ▶ Add** to start the **Satellite Assembly Wizard** or **Resource DLL Wizard**.
2. Specify the directory path of the old resource module in the appropriate dialog.
3. In the **Update Mode** column, select **Update**. If a resource module already exists for the language (in the directory you have specified), click in the **Update Mode** column to select **Update** or **Overwrite**. Choose **Update** to keep and modify the existing assembly project. Choose **Overwrite** to create a new, empty project and to delete the old project and any translations it contains.
4. Click **Finish**.

See Also

Localizing Applications (see page 18)

Editing Resource Files in the Translation Manager (see page 170)

Setting Up the External Translation Manager (see page 172)

2.6.2 Editing Resource Files in the Translation Manager

After you have added languages to your project by using the **Satellite Assembly Wizard** or **Resource DLL Wizard**, you can use the Translation Manager to view and edit your resource files. You can edit resource strings directly, add translated strings to the Translation Repository, or get strings from the Translation Repository.

To edit resource strings

1. Open a project that includes languages.
2. Choose **View ▶ Translation Manager ▶ Translation Editor**.
3. Expand the project tree view to display the resource files that you want to edit.

   Tip: Use the expand and collapse icons on the toolbar above the tree view.

4. Click the resource file you want to edit. The resource strings in the file are displayed in a grid in the right pane.
5. Click the field that you want to edit and type the new text directly in the grid, right-click the field and choose Edit to edit the string in a dialog box, or click the **Multi-line editor** icon on the toolbar above the grid.
6. Optionally, enter a comment in the **Comment** field.
7. Optionally, set the translation status for the string by using the drop-down list in the **Status** field.
8. Click the **Save Translation** icon on the toolbar above the grid to update the resource file.

   Tip: To display the original form or translated form, click the Show original form

   and **Show translated form** icons in the toolbar above the grid.

To add a resource string to the Translation Repository

1. After editing a resource string in the Translation Manager, right-click the string that you want to add to the Translation Repository.
2. Choose Repository ▸ Add strings to repository. The resource string is added to the Translation Repository and can be viewed by closing the Translation Manager and choosing Tools ▸ Translation Repository.

To get a resource string from the Translation Repository

1. In the Translation Manager, click the **Workspace** tab.
2. Expand the project tree view to display the resource files that you want to edit. The `.resx` files are listed under the **.NET Resources** node. The `.nfm` files are listed under the **Forms** node.
3. Click the resource file you want to edit. The resource strings in the file are displayed in a grid in the right pane.
4. Right-click the field that you want to update and choose Repository ▸ Get strings from repository. If the Translation Repository contains only one translation that matches the selected source string, it copies that translation into the target language column. If the Repository contains more than one match for the selected resource, its default behavior is to retrieve the first matching translation it finds.

   Tip: To change this behavior, close the Transaction Manager and choose Tools->Translation Tools Options

   , click the **Repository** tab, and change the **Multiple Find Action** setting.

To open the resource file in a text editor

1. In the Translation Manager, click the **Project** tab.
2. Click the **Files** tab.
3. Double-click the resource file that you want to update. The file opens in a text editor.
4. Change the file as needed and save it.

   Tip: To change the text editor used by the Translation Manager, choose Tools->Translation Tools Options

   and change executable file specified in the **External Editor** field.

See Also

Localizing Applications (see page 18)

Adding Languages to a Project (see page 169)

2.6.3 Setting the Active Language for a Project

After adding languages to your project with the **Satellite Assembly Wizard** or the **Resource DLL Wizard**, the base language module is loaded when you choose **Run**▶**Run**. However, you can load a different language module by setting the active language for the project.

To set the active language

1. In the IDE, recompile the resource module for the language you want to use.
2. Choose **Project**▶**Languages**▶**Set Active**. The **Set Active Language** wizard displays a list of the languages in the project. The base language appears in angle brackets at the top of the language list, for example, `<English (United States)>`.
3. Select a language from the list and click **Finish**.

See Also

Localizing Applications (see page 18)

Adding Languages to a Project (see page 169)

2.6.4 Setting Up the External Translation Manager

If you do not have the RAD Studio IDE, you can use the External Translation Manager (ETM) to localize an application. To use ETM, the developer must provide you with the required ETM files and project files.

Note: The Microsoft .NET Framework must be installed on your computer before you install ETM.

To set up and register the ETM files

1. Obtain the following ETM files from the developer. By default these files are in either the `Program Files\CodeGear\RAD Studio\5.0\Bin` or the `Windows\system32` directory on the developer's computer.

   Note: If the developer chose to install only the Delphi for Win32 personality of RAD Studio, the files marked with an asterisk (*) will not be available on the developer's computer.

```
Borland.Delphi.dll *
Borland.Globalization.dll *
Borland.ITE.dll *
Borland.ITE.FormDesigner.dll *
Borland.SCI2.dll *
Borland.Vcl.dll *
Borland.VclRtl.dll *
Borland.VclX.dll *
designide100.bpl
dfm100.bpl
DotnetCoreAssemblies100.bpl *
etm.exe
IDECtrls100.bpl
itecore100.bpl
itedotnet100.bpl *
rc100.bpl
ResX100.bpl *
rtl00.bpl
vclide100.bpl
xmlrtl100.bpl
```

2. Create a directory, such as `C:\ETM`.
3. Copy the ETM files from the developer into the directory.
4. Open ETM. From Windows Explorer, double-click `etm.exe`. From the command line, enter `etm.exe.`
5. Choose **Tools**▶**Options**▶**Packages**.
6. Click the **Add** button to display the **Open** dialog box.

7. Navigate to the directory that contains the ETM files. Make sure that the **Files of type** filter is set to **Designtime packages (dcl*.bpl)**.
8. Select all of the designtime packages in the directory and click **OK**.

The designtime packages are registered and you can now begin using ETM.

To set up the project to be translated

1. Obtain a zipped translation kit of the project to be translated from the developer. The kit should include the following:

- a satellite assembly or resource DLL for each language to be translated
- the `.dproj` project file generated by using **File**▶**Save as** in the ETM project
- the standalone translation repository (`*.tmx`) files

2. Unzip the translation kit into a directory of your choice.

See Also

Localizing Applications (see page 18)

Adding Languages to a Project (see page 169)

Editing Resource Files in the Translation Manager (see page 170)

2.6.5 Updating Resource Modules

When you add an additional resource, such as a button on a form, you must update your resource modules to reflect your changes.

To update resource modules

1. Save and build your project. If you are using the ETM, reopen the saved project.
2. Update the resource modules:

- In the IDE, choose **Project**▶**Languages**▶**Update Localized Projects**.
- In ETM, choose **Project**▶**Run Updaters** (or press F9) or click the **Files** tab and then click the **Run Updaters** button (F9).

3. After updating in the internal Translation Manager, rebuild each resource module project by opening the project in the IDE and choosing **Project**▶**Compile**.

   Tip: To simplify this process, you can maintain all the projects, along with the application itself, in a single project group that can be compiled from the Project Manager

   by choosing **Project**▶**Compile All**.

See Also

Adding Languages to a Project (see page 169)

Editing Resource Files in the Translation Manager (see page 170)

Setting Up the External Translation Manager (see page 172)

2.6.6 Using the External Translation Manager

Translators who do not have the RAD Studio IDE can use the External Translation Manager (ETM) instead of the Translation

Manager. The steps for using the ETM are similar to those for the internal Translation Manager.

Note: ETM must be set up and operational on your computer before using the following procedure. See in the link listed at the end of this topic for details.

To run the ETM

1. To run the ETM from the command line, enter: `etm.exe [files]` where `[files]` is the optional project group file or the project files.
2. To run the ETM from Windows Explorer, double-click `etm.exe`

To localize an application using the ETM

1. In ETM, choose **File ▶ Open** and open the project to be translated.
2. Click the **Workspace** tab.
3. Expand the project tree view to display the resource files that you want to edit.

   Tip: Use the expand and collapse icons on the toolbar above the tree view.

4. Click the unit file that you want to edit. The resource strings in the file are displayed in a grid in the right pane.
5. Click the field that you want to edit and do one of the following:

- type the new text directly in the grid
- right-click the field and choose Edit to edit the string in a dialog box
- click the **Multi-line editor** icon on the toolbar above the grid

6. Optionally, enter a comment in the **Comment** field.
7. Optionally, set the translation status for the string by using the drop-down list in the **Status** field.
8. Click the **Save Translation** icon on the toolbar above the grid to update the resource file.

After you have finished the translations, you can send the translated files back to the developer to add to the project.

To remove languages from your project

1. Open your project.
2. On the **Languages** tab, uncheck the check box for the language you want to remove.
3. Click the **Files** tab and click the **Run Updaters** button.

ETM removes the selected assemblies or DLLs from your project, but it does not delete them, the source of them, or the directories they reside in.

See Also

Setting Up the External Translation Manager (↗ see page 172)

2.7 Managing Memory

This section provides how-to information on using the Memory Manager, covering how to configure the Memory Manager, increase the memory address space, monitor the Memory Manager, use the memory map, share memory, and report and manage memory leaks.

Topics

Name	Description
Configuring the Memory Manager (↗ see page 175)	This section describes how to configure the Memory Manager. **Note:** You can change some memory manager configuration settings while the Memory Manager is in use. All the configuration settings are global settings and affect all threads that are using the Memory Manager. Unless otherwise stated, all functions and procedures are thread safe. These configuration options are for the local Memory Manager only. Setting these options inside a library when the library is sharing the Memory Manager of the main application will have no effect.
Increasing the Memory Address Space (↗ see page 176)	This section describes how to extend the address space of the Memory Manager beyond 2 GB. **Note:** The default size of the user mode address space for a Win32 application is 2GB, but this can optionally be increased to 3GB on 32-bit Windows and 4GB on 64-bit Windows. The address space is always somewhat fragmented, so it is unlikely that a GetMem request for a single contiguous block much larger than 1GB will succeed – even with a 4GB address space.
Monitoring Memory Usage (↗ see page 177)	This section describes how to monitor the state of the Memory Manager. The Memory Manager provides two procedures that allow the application to monitor its own memory usage and the state of the process' address space. Both functions are thread safe.
Registering Memory Leaks (↗ see page 178)	This section describes how to register and unregister expected memory leaks. When you allocate memory that you don't expect to free, you can register it with the Memory Manager. The Memory Manager adds it to a list of areas to ignore when it checks for memory leaks. When you unregister a memory location, the Memory Manager removes it from its list of expected memory leaks.
Sharing Memory (↗ see page 178)	This section describes how to share memory using the Memory Manager. On Win32, if a DLL exports routines that pass long strings or dynamic arrays as parameters or function results (whether directly or nested in records or objects), then the DLL and its client applications (or DLLs) must all share the same memory manager. The same is true if one application or DLL allocates memory with **New** or **GetMem** which is deallocated by a call to **Dispose** or FreeMem in another module. There are two mutually exclusive methods through which the Memory Manager can be shared between an application and... more (↗ see page 178)

2.7.1 Configuring the Memory Manager

This section describes how to configure the Memory Manager.

Note: You can change some memory manager configuration settings while the Memory Manager is in use. All the configuration settings are global settings and affect all threads that are using the Memory Manager. Unless otherwise stated, all functions and procedures are thread safe.

These configuration options are for the local Memory Manager only. Setting these options inside a library when the library is sharing the Memory Manager of the main application will have no effect.

To set the minimum block alignment for the Memory Manager

1. Use the function GetMinimumBlockAlignment to fetch the current minimum block alignment.

2. Select the appropriate memory block alignment for your application. Available block alignments are 8-byte (mba8byte) and 16-byte (mba16byte).
3. To change the memory block alignment, use the procedure SetMinimumBlockAlignment.

   Note: Memory allocated through the Memory Manager is guaranteed to be aligned to at least 8-byte boundaries. 16-byte alignment is useful when memory blocks will be manipulated using SSE instructions, but may increase the memory usage overhead.

To report memory leaks on shutdown

1. Set the global variable ReportMemoryLeaksOnShutdown to **True**.
2. When the Memory Manager shuts down, it scans the memory pool and report all unregistered memory leaks in a message dialog. To register and unregister expected memory leaks, use the RegisterExpectedMemoryLeak and UnregisterExpectedMemoryLeak procedures.

   Note: The Memory Manager can report memory that was allocated but not freed at the time the Memory Manager shuts down. Such memory blocks are called *memory leaks* and are often the result of programming errors. The default value for ReportMemoryLeaksOnShutdown is False

   .

   The class of a leak is determined by examining the first dword in the block. The reported classes of leaks may not always be 100% accurate. A leak is reported as a string leak if it appears to be an **AnsiString**. If the Memory Manager is unable to estimate the type of leak, it will be reported as belonging to the unknown class.

To never sleep on thread contention in the Memory Manager

1. Set the global variable NeverSleepOnMMThreadContention to **True**.
2. When a thread contention occurs inside the Memory Manager, it will wait inside a loop until the contention is resolved.

   Note: The Memory Manager is a shared resource, and when many threads in the application attempt to perform a Memory Manager operation at the same time, one or more threads might have to wait for a pending operation in another thread to complete before it can continue. This situation is called *thread contention*. When a thread contention occurs inside the Memory Manager, the default behavior is to relinquish the remaining time in the thread’s time slice. If the resource is still not available when the thread enters its next time slice, the Memory Manager calls the OS procedure Sleep to force it to wait longer (roughly 20 milliseconds) before trying again.

   This behavior works well on machines with single or dual core CPUs, and also when the ratio of the number of running threads to number of CPU cores is relatively high (greater than 2:1). In other situations, better performance can be obtained by entering a busy waiting loop until the resource becomes available. If NeverSleepOnMMThreadContention is True

   , the Memory Manager will enter a wait loop instead of scheduling out. The default value for NeverSleepOnMMThreadContention is **False**.

See Also

Memory Management (see page 644)

Increasing the Memory Manager Address Space Beyond 2GB (see page 176)

Registering Memory Leaks (see page 178)

Monitoring the Memory Manager (see page 177)

Sharing Memory (see page 178)

2.7.2 Increasing the Memory Address Space

This section describes how to extend the address space of the Memory Manager beyond 2 GB.

Note: The default size of the user mode address space for a Win32 application is 2GB, but this can optionally be increased to

3GB on 32-bit Windows and 4GB on 64-bit Windows. The address space is always somewhat fragmented, so it is unlikely that a GetMem request for a single contiguous block much larger than 1GB will succeed – even with a 4GB address space.

To enable and use a larger address space

1. Make sure the operating system supports a larger address space. A user mode address space larger than 2GB is supported by 64-bit editions of Windows, as well as 32-bit editions that support the /3GB option in `boot.ini` (and have it set).
2. Set the appropriate linker directive. The operating system must be informed through a flag in the executable file header that the application supports a user mode address space larger than 2GB, otherwise it will be provided with only 2GB. To set this flag, specify `{$SetPEFlags IMAGE_FILE_LARGE_ADDRESS_AWARE}` in the .dpr file of the application.
3. Make sure that all libraries and third party components support the larger address space. With a 2GB address space the high bit of all pointers is always 0, so a larger address space may expose pointer arithmetic bugs that did not previously show any symptoms. Such bugs are typically caused when pointers are typecast to integers instead of cardinals when doing pointer arithmetic or comparisons.

   Note: Memory allocated through the Memory Manager is guaranteed to be aligned to at least 8-byte boundaries. 16-byte alignment is useful when memory blocks will be manipulated using SSE instructions, but may increase the memory usage overhead. The guaranteed minimum block alignment for future allocations can be set with SetMinimumBlockAlignment.

See Also

Memory Management (↗ see page 644)

Configuring the Memory Manager (↗ see page 175)

Registering Memory Leaks (↗ see page 178)

Monitoring the Memory Manager (↗ see page 177)

Sharing Memory (↗ see page 178)

2.7.3 Monitoring Memory Usage

This section describes how to monitor the state of the Memory Manager.

The Memory Manager provides two procedures that allow the application to monitor its own memory usage and the state of the process' address space. Both functions are thread safe.

To monitor memory usage for your application:

1. Call the procedure GetMemoryManagerState.
2. Inspect the populated TMemoryManagerState structure and extract the needed Memory Manager state information. The structure has fields detailing the total number of allocations, the sum of their sizes, as well as the total reserved address space. The statistics are subdivided into three categories: small, medium and large allocations.

To get a map of the memory address space for a process

1. Call the procedure GetMemoryMap.
2. Inspect the populated TMemoryMap array and extract the needed information regarding the process' address space. The array contains a TChunkStatus entry for every possible 64K block in the process' address space.

See Also

Memory Management (↗ see page 644)

Configuring the Memory Manager (↗ see page 175)

Increasing the Memory Manager Address Space Beyond 2GB (↗ see page 176)

Registering Memory Leaks (↗ see page 178)

Sharing Memory (↗ see page 178)

2.7.4 Registering Memory Leaks

This section describes how to register and unregister expected memory leaks.

When you allocate memory that you don't expect to free, you can register it with the Memory Manager. The Memory Manager adds it to a list of areas to ignore when it checks for memory leaks. When you unregister a memory location, the Memory Manager removes it from its list of expected memory leaks.

To register an expected memory leak:

1. Identify the pointer to the memory area you don't expect to free.
2. Pass the pointer to RegisterExpectedMemoryLeak.

To unregister an expected memory leak

1. Identify the pointer to the memory area you want to unregister.
2. Pass the pointer to UnregisterExpectedMemoryLeak.

See Also

Memory Management (↗ see page 644)

Configuring the Memory Manager (↗ see page 175)

Increasing the Memory Manager Address Space Beyond 2GB (↗ see page 176)

Monitoring the Memory Manager (↗ see page 177)

Sharing Memory (↗ see page 178)

2.7.5 Sharing Memory

This section describes how to share memory using the Memory Manager. On Win32, if a DLL exports routines that pass long strings or dynamic arrays as parameters or function results (whether directly or nested in records or objects), then the DLL and its client applications (or DLLs) must all share the same memory manager. The same is true if one application or DLL allocates memory with **New** or **GetMem** which is deallocated by a call to **Dispose** or FreeMem in another module. There are two mutually exclusive methods through which the Memory Manager can be shared between an application and its libraries: **ShareMem** and **SimpleShareMem**.

Note: When a DLL is statically linked to an application, the DLL is initialized before the application. The application will use the memory manager of the DLL if the SimpleShareMem sharing method is used in both. Only the module that is sharing its memory manager can change memory manager settings and retrieve memory manager statistics. Changing settings in any of the other modules will have no effect, since their memory managers are not used.

It is possible, but rarely needed, to control the memory manager sharing mechanism manually.

To use ShareMem

1. List ShareMem as the first unit in the program and library uses clause. Your modules will become dependant on the external

BORLNDMM.DLL library, allowing them to share dynamically allocated memory.

2. Deploy *BORLNDMM.DLL* with your application or DLL that uses ShareMem. When an application or DLL uses ShareMem, its memory manager is replaced by the memory manager in *BORLNDMM.DLL*.

To use SimpleShareMem

1. List SimpleShareMem as the first unit in the program and library uses clause in each of your modules. The module that is initialized first will be the module that will share its memory manager. All modules initialized after that will use the memory manager of the first module.
2. The module that is initialized first will be the module that will share its memory manager. All modules initialized after that will use the memory manager of the first module.

See Also

Memory Management (↗ see page 644)

Configuring the Memory Manager (↗ see page 175)

Registering Memory Leaks (↗ see page 178)

Increasing the Memory Manager Address Space Beyond 2GB (↗ see page 176)

Monitoring the Memory Manager (↗ see page 177)

2.8 Unit Test Procedures

This section provides how-to information on using the features of DUnit and NUnit.

Topics

Name	Description
Developing Tests (see page 180)	The structure of a unit test largely depends on the functionality of the class and method you are testing. The Unit Test Wizards generate skeleton templates for the test project, `setup` and `teardown` methods, and basic test cases. You can then modify the templates, adding the specific test logic to test your particular methods. The following describes the procedures for creating a Unit Test Project and a Unit Test Case. Follow these procedures in the order shown. You must create the Unit Test Project prior to creating the associated test cases. The **Unit Test Case Wizard** is available only if... more (see page 180)

2.8.1 Developing Tests

The structure of a unit test largely depends on the functionality of the class and method you are testing. The Unit Test Wizards generate skeleton templates for the test project, `setup` and `teardown` methods, and basic test cases. You can then modify the templates, adding the specific test logic to test your particular methods.

The following describes the procedures for creating a Unit Test Project and a Unit Test Case. Follow these procedures in the order shown. You must create the Unit Test Project prior to creating the associated test cases. The **Unit Test Case Wizard** is available only if the current active project is a Unit Test Project.

To create a test project

1. Choose **File ▶ New ▶ Other**.
2. Open the **Unit Test** folder.
3. Double-click the **Test Project** gallery item. This starts the **Test Project Wizard** and displays the **Specify Test Project Details** page.
4. Fill in the appropriate details or accept the defaults. Enter the following:

- **Project name:** Enter the name for the new test project, or accept the default. The default is the name of the active project with the word *Tests* appended to the name. If there is no active project, the default will be *UnitTest* with a sequence number appended.
- **Location:** Enter the full pathname for the folder in which to create the test project, or accept the default. The default is a subfolder named *test* under the active project folder. If there is no active project, then the default is the default project folder. You can click the ellipsis (**...**) to display a **Browse** dialog box from which you can select the location.
- **Personality:** Select the personality (code language) from the drop down list, or accept the default. The default is the personality of the active project.

5. If you do not want the test project added to your project group, uncheck the **Add to Project Group** check box.
6. Click **Next** to proceed, or click **Finish** to accept the remaining defaults. If the **Finish** button is available, you can click it at any point to accept the default values for any remaining fields and immediately generate the new test project. Otherwise, click **Next** to proceed to the **Specify Test Framework Options** page.
7. Fill in the appropriate details or accept the defaults. Enter the following:

- **Test Framework:** For the Delphi.Net personality, you can choose either **DUnit** or **NUnit** from the drop down list. For the

Delphi and C++ personalities, only the DUnit framework is supported, so you cannot change this value. For C#, only the NUnit framework is supported.

- **Test Runner:** Choose either **GUI** or **Console** from the drop down list. The Console Test Runner directs output to the console. The GUI Test Runner displays the results interactively in a GUI window, with results color-coded to indicate success or failure.

8. Click **Finish**. The **Test Project Wizard** generates the test project template. For the Delphi and C# personalities, this also adds the necessary source references into the test project template; you can skip the next step. For C++Builder, you must manually link the test project with the C++ classes you want to test; proceed to the next step.
9. **For C++Builder, only:** Link the new test project to the source to be tested. For C++Builder, you must manually link the test project with the C++ code you want to test. You can use any of the following techniques to do this, as appropriate:

- Add the C++ code directly to the test project.
- Add an .obj file to the test project.
- Add a .lib file (either a static library or an import library for a DLL) to the test project.
- Include the header file that implements the class you want to test. See the next section, **Adding files to a test project**, for instructions.

Adding files to a test project

1. Open the test project file and activate it. To activate the file, select the file in the **Project Manager** and click the **Activate** button.
2. In the **Project Manager**, right-click on the test project name. This displays a pop-up menu of project operations.
3. Choose Add.... This displays the **Add to Project** file browser, from which you can select the file to include in your test project.
4. Select the file to add and click **OK**. This adds the select file to the test project.

To create a test case

1. Click the **Code** tab for the file containing the classes you want to test. This makes the file active in the **Code Editor**.
2. Choose **File** ▶ **New** ▶ **Other**.
3. Open the **Unit Test** folder.
4. Double-click the **Test Case** gallery item. This starts the **Test Case Wizard** and displays the **Select Methods to Test** page.
5. Enter the name of the source file that contains the classes you want to test. You can click the ellipsis (**...**) to display an **Open** dialog box, from which you can select the file.
6. Select the classes and methods for which you want to create tests. In the **Available classes and methods** list, click the check box next to an item to select or deselect it. By default, all classes and methods are selected. You can deselect individual methods in the list. The wizard generates test case templates for the checked (selected) methods only. If you deselect a class, the wizard ignores the entire class and all of its methods, even if you do not deselect the methods. If you select a class but do not select any methods in that class, the wizard generates a test case for the class, but does not generate any test methods for that class.
7. Click **Next** to proceed, or click **Finish** to accept the remaining defaults. If the **Finish** button is available, you can click it at any point to accept the default values for any remaining fields and immediately generate the new test case. Otherwise, click **Next** to proceed to the **Specify Test Case Details** page.
8. Fill in the appropriate details or accept the defaults.

- **Test Project:** Select the test project from the drop down list. The default is the active test project; if you just created a test project using the **Test Project Wizard**, then the new test project is the default.
- **File name:** Enter a filename for the test case you are creating or accept the default. The default is the name of the source file to be tested, with the prefix *Test* added to the name.
- **Test Framework:** For the Delphi for .Net personality, you can choose either **DUnit** or **NUnit** from the drop down list. For the Delphi for Win32 and C++ personalities, only the DUnit framework is supported, so you cannot change this value. For C#, only the NUnit framework is supported.
- **Base Class:** Select the base class from the drop down list or accept the default. The default is **TTestCase**, which is the

2

default base **TestCase** class. In most cases, you can use the default. However, you can specify a custom **TTestCase** class that you have created. In addition, if you are testing a hierarchy of objects, you can derive the new test case from the **TestCase** class of a base object of the object being tested. This allows a derived class to inherit a test created for the base type for that class.

9. Click **Finish**. The wizard generates a test case file with the name you specified.

To write a test case

1. Add code to the `SetUp` and `TearDown` methods in the test case template(s), if needed.
2. Add asserts to the test methods.

To run a test case in the GUI Test Runner

1. Activate the file containing the classes you want to run. Select the file in the **Project Manager** and then click the **Activate** button.
2. Choose **Run**▶**Run**. The **GUI Test Runner** starts up immediately on execution of your application.
3. Select one or more tests from the tests list.
4. Click the **Run** button. The test results appear in the **Test Results** window. Any test highlighted with a green bar passed successfully. Any test highlighted in red failed. Any test highlighted in yellow was skipped.
5. Review the test results.
6. Fix the bugs and rerun the tests.

See Also

Unit Testing Overview (↗ see page 70)

DUnit Overview (↗ see page 72)

NUnit Overview (↗ see page 76)

2.9 Together Procedures

This section provides how-to information on using the Together features.

Topics

Name	Description
Configuring Together (↗ see page 183)	Together is flexibly configurable. Use the **Options** dialog window to tune modeling features to best fit your requirements. The **Options** dialog window provides a number of diagram customization settings. You can configure the appearance and layout of the diagrams, specify font properties, member format, and level of detail.
Together Refactoring Procedures (↗ see page 184)	This section provides how-to information on using Together refactoring facilities.
Opening the UML 2.0 Sample Project (↗ see page 190)	
Together Diagram Procedures (↗ see page 190)	This section provides how-to information on using Together UML diagrams.
Together Documentation Generation Procedures (↗ see page 247)	This section provides how-to information on using Together Documentation Generation facilities.
Using Online Help (↗ see page 248)	
Together Object Constraint Language (OCL) Procedures (↗ see page 248)	This section provides how-to information on using Together OCL facilities.
Working with a Namespace or a Package (↗ see page 250)	Namespaces are used in implementation projects, and packages in design projects.
Together Pattern Procedures (↗ see page 251)	This section provides how-to information on using patterns with Together.
Together Project Procedures (↗ see page 262)	This section provides how-to information on using Together projects.
Together Quality Assurance Procedures (↗ see page 271)	This section provides how-to information on using Together Quality Assurance facilities.

2.9.1 Configuring Together

Together is flexibly configurable. Use the **Options** dialog window to tune modeling features to best fit your requirements.

The **Options** dialog window provides a number of diagram customization settings. You can configure the appearance and layout of the diagrams, specify font properties, member format, and level of detail.

To configure Together settings:

1. On the main menu, choose **Tools** ▶ **Options**.
2. In the **Options** dialog window, expand the Together category.
3. Select the desired option level.
4. For the Project and Diagram option levels, choose the project or diagram where the configuration changes should apply. To do that, click the chooser buttons in the corresponding fields and select the desired project or diagram from the model.
5. Click the desired subcategory.
6. Edit configuration options as required.
7. Click **OK** to apply changes and close the dialog window.

You can make configuration options final at a certain parent level and disable any changes on the lower levels:

To disable configuration changes:

1. On the main menu, choose **Tools** ▶ **Options**.
2. Click the Together category to expand it.

3. Select the required sub-category (default, project group or project).
4. Check the **Disable sublevels** option.

See Also

Option Levels (see page 1088)

Option Value Editors (see page 1101)

2.9.2 Together Refactoring Procedures

This section provides how-to information on using Together refactoring facilities.

Topics

Name	Description
Refactoring: Changing Parameters (see page 184)	
Refactoring: Extracting Interfaces (see page 185)	The following conditions should be met for extracting interfaces: • Only non-static methods can be extracted. • All methods in the extracted interface are public. • If the name specified for the new interface coincides with the name of an existing interface in the same namespace, all the methods will be extracted into an existing interface.
Refactoring: Extracting Method (see page 185)	
Refactoring: Extracting Superclass (see page 186)	
Refactoring: Creating Inline Variables (see page 186)	
Refactoring: Introducing Fields (see page 187)	
Refactoring: Introducing Variables (see page 187)	
Refactoring: Moving Members (see page 188)	
Refactoring: “Pull Members Up" and “Push Members Down” (see page 188)	Moving members assumes that the member is either moved to the target location being deleted from the original location, or created in the target location being preserved on the original one.
Refactoring: Renaming Elements (see page 189)	To rename a local variable or parameter, right-click a variable name in the source code and choose **Refactoring** ▶**Rename** on the main menu. For the other code elements, you can use the source-code Editor, the **Diagram View**, the Model View, or the Refactoring main menu. **Tip:** In order to make renaming overloads possible, the method should have its override property set to `true`.
Refactoring: "Safe Delete" (see page 189)	

2.9.2.1 Refactoring: Changing Parameters

To change parameters, follow these steps:

1. Select method in the **Diagram View**, in the **Model View** or in the Editor.
2. Choose Refactoring->Change Parameters from the main menu.

   Tip: Alternatively, you can right-click and choose Refactoring->Change Parameters on the context menu.

3. In the resulting dialog, select parameter from the list and choose the desired action:

- To add a new parameter, click **Add**, and specify the parameter name, type and default value.
- To delete parameter, click **Remove**.

- To rename parameter, click the **Name** field, and edit the parameter name using the in-place editor.

4. If applicable, check **Refactor Ancestors.**
5. Check **Preview Usages** if necessary.

- If this option is checked when you click OK, the **Refactoring** window opens allowing you to review the refactoring before committing to it. Click the **Perform refactoring** button to complete the changes. You can use the Undo and Redo commands as necessary once you have performed the refactoring.
- If this option is cleared when you click OK, the **Refactoring** window opens with the change completed. You can use the Undo and Redo commands as necessary once you have performed the refactoring.

See Also

Refactoring overview (see page 98)

Change Parameters dialog box (see page 961)

2.9.2.2 Refactoring: Extracting Interfaces

The following conditions should be met for extracting interfaces:

- Only non-static methods can be extracted.
- All methods in the extracted interface are public.
- If the name specified for the new interface coincides with the name of an existing interface in the same namespace, all the methods will be extracted into an existing interface.

To extract an interface:

1. Select one or more code elements (class, interface, field, method, event, property, or indexer) in the **Diagram View** or **Model View**.
2. On the main menu, chooseRefactoring ▶ Extract Interface

   Tip: Alternatively, you can choose Refactoring->Extract Interface

   on the context menu of the selection.
3. In the **Extract interface** dialog box, enter the name for the interface and designate its namespace, if applicable.
4. Specify the members to be used in the resulting interface by setting or clearing the respective check-boxes.
5. Click **OK**. The **Refactoring** window opens allowing you to review the refactoring before committing to it.
6. Click the **Perform refactoring** button to complete the extraction.

See Also

Refactoring Overview (see page 98)

Refactoring Operations (see page 1115)

2.9.2.3 Refactoring: Extracting Method

To extract a method:

1. In the Editor, open the class or interface containing the code fragment that you wish to extract.
2. Place the mouse cursor in the desired fragment of source code. Refactoring determines the beginning and the end of the relevant statement.
3. On the main menu, choose Refactoring ▶ Extract Method

   Tip: Alternatively, right-click the code fragment and choose Refactoring->Extract Method on the context menu.

4. In the dialog box that opens, specify the following information:

- Name of the new method
- Visibility (public, protected, private, internal, internal protected)
- Header comment
- Whether the method is Static.

5. Click **OK** to complete the extraction and create the new method.

Tip:

- When applying Extract Method, parameters and local variables in the selected code fragment become the parameters of the new method.
- The code fragment cannot contain a return statement of the original method. An error message displays if you attempt to include a return statement in the code fragment.
- The code fragment cannot modify more than one single local variable. An error message displays if you violate this restriction.
- If the selected code fragment is repeated in several locations, it is your responsibility to replace these fragments in the appropriate locations with the proper method calls.

See Also

Refactoring overview (see page 98)

2.9.2.4 Refactoring: Extracting Superclass

To use the "Extract superclass" operation:

1. Select one or more code elements (class, interface, field, method, event, property, or indexer) in the Diagram or **Model View**.
2. On the main menu, choose**Refactoring**▶**Extract Superclass**

   Tip: Alternatively, you can choose Refactoring->Extract Superclass

   on the context menu of the selection.

3. In the **Extract superclass** dialog box, enter the name for the interface and designate its namespace, if applicable.
4. Specify the members to be used in the resulting superclass by setting or clearing the respective check-boxes. If applicable, indicate that a method is abstract in the extracted superclass.
5. Click **OK**. The **Refactoring** window opens allowing you to review the refactoring before committing to it.
6. Click the **Perform refactoring** button to complete the extraction.

See Also

Refactoring overview (see page 98)

Refactoring Operations (see page 1115)

2.9.2.5 Refactoring: Creating Inline Variables

To create an inline variable:

1. Select the local variable in the Editor.
2. On the main menu, choose**Refactoring**▶**Inline variable**

   Tip: Alternatively, you can choose Refactoring->Inline variable

   on the context menu. The resulting dialog reports the number of variable occurrences that the Inline Variable command will be applied to.

3. Click OK to complete refactoring.

   Warning: The variable that you select for creating an inline variable, should not be updated later in the source code. If it is, the following error message will display: "Variable index is accessed for writing."

   For example, if you use the Inline Variable refactoring command on the local variable, `index`, shown below:

```
public void findIndex() {
                        int index = 2;
                        System.Console.Writeline("Index is: {0}", index);
                }
```

then the following refactoring occurs:

```
public void findIndex() {
   System.Console.Writeline("Index is: {0}", 2);
}
```

See Also

Refactoring overview (↗ see page 98)

Inline Variable dialog box (↗ see page 967)

2.9.2.6 Refactoring: Introducing Fields

To introduce a field:

1. Select expression in the Editor.
2. On the main menu, chooseRefactoring ▸ Introduce Field

   Tip: Alternatively, you can choose Refactoring->Introduce Field

   on the context menu.
3. In the resulting dialog, specify the following:

- **Name:** Enter the name of the new field
- **Visibility**: Using the list box, choose the visibility for the new field from *public, protected, private, internal,* or *internal protected*.
- **Initialize**: Choose where to initialize the new field. Using the list box, choose from *Current method, Class constructor(s)*, or *Field declaration*.

4. If applicable, check the Static and Replace all occurrences fields.
5. Click **OK** to complete the refactoring.

See Also

Refactoring overview (↗ see page 98)

Introduce Field dialog box (↗ see page 968)

2.9.2.7 Refactoring: Introducing Variables

To introduce a new variable:

1. Select variable in the Editor.
2. On the main menu, chooseRefactoring ▸ Introduce Variable

   Tip: Alternatively, you can choose Refactoring->Introduce Variable on the context menu.

3. In the resulting dialog, specify the Name of the new variable. The new variable created is given the same type as the original variable.
4. If desired, check **Replace all occurrences**. The **Introduce Variable** dialog indicates the number of occurrences that it will replace with the new variable.

   Note: The refactoring does not replace any occurrences of the variable prior to the point in the code at which you selected to introduce the new variable.

See Also

Refactoring overview (↗ see page 98)

Introduce variable dialog box (↗ see page 968)

2.9.2.8 Refactoring: Moving Members

To move a static member to a different class:

1. Select one or more static members in the **Diagram View** or **Model View**.
2. On the main menu choose **Refactoring ▶ Move**

   Tip: Alternatively, right-click on the selection and choose Refactoring->Move Members

   on the context menu
3. In the **Move Members** dialog, use the **Move Members** field to select which static members to move. You can deselect/select the static members by clearing/checking the check box next to the name of the member
4. Use the **To** field to enter the fully-qualified name for the target class where the selected code element or elements will reside.
5. Click **OK**.

See Also

Refactoring overview (↗ see page 98)

Move Members dialog box (↗ see page 969)

2.9.2.9 Refactoring: “Pull Members Up" and “Push Members Down”

Moving members assumes that the member is either moved to the target location being deleted from the original location, or created in the target location being preserved on the original one.

To move a member:

1. Select member in the **Diagram View** or in the **Model View**.

   Tip: In the editor, place the mouse cursor on the member name.

2. Choose **Refactoring ▶ Pull Members Up/Push Members Down** on the context menu or on the main menu.
3. In the resulting dialog box, specify additional information required to make the move.

- In the top pane of the dialog box, check the members to be moved.
- In the bottom pane of the dialog box, that shows the class hierarchy tree, select the target class.

4. Click **OK**.

5. In the **Refactoring** window that opens, review the refactoring before committing to it. Click the **Perform refactoring** button to complete the move.

   Tip: Moving members is more complicated than moving classes among namespaces, because class members often contain references to each other. A warning message is issued when Pull Members Up or Push Members Down has the potential for corrupting the syntax if the member being moved references other class members. You can choose to move the class member and correct the resulting code manually.

See Also

Refactoring overview (↗ see page 98)

2.9.2.10 Refactoring: Renaming Elements

To rename a local variable or parameter, right-click a variable name in the source code and choose Refactoring ▶ Rename on the main menu. For the other code elements, you can use the source-code Editor, the **Diagram View**, the Model View, or the Refactoring main menu.

Tip: In order to make renaming overloads possible, the method should have its override

property set to `true`.

To rename an element:

1. Select element in the **Diagram View** or in the **Model View**.

   Tip: In the editor, place the mouse cursor on the element name.

2. Choose Refactoring ▶ Rename on the context menu or on the main menu.
3. In the resulting dialog box, specify new name of the element.
4. Click **OK**.
5. In the **Refactoring** window that opens, review the refactoring before committing to it. Click the **Perform refactoring** button to complete the move.

See Also

Refactoring overview (↗ see page 98)

Rename dialog box (↗ see page 975)

2.9.2.11 Refactoring: "Safe Delete"

To safely delete an element:

1. Select the element to be deleted.
2. On the main menu, choose Refactoring ▶ Safe Delete

   Tip: Alternatively, right-click on the element and choose Refactoring->Safe Delete

   on the element's context menu.

3. In the **Safe Delete** dialog box that reports the element to delete and any usages of that element:

- If no usages are found, press **Delete**.
- If usages are found, click **View usages**. The Refactoring window opens allowing you to review the refactoring before committing to it. Click the **Perform refactoring** button to delete the element.

See Also

Refactoring overview (see page 98)

Safe Delete dialog box (see page 975)

2.9.3 Opening the UML 2.0 Sample Project

To open the UML 2.0 sample project:

1. From the File menu, select Open | Project. The Open Project dialog box opens.
2. Navigate to UML 2.0 Samples Project in `C:\Documents and Settings\%user_name%\My Documents\RAD Studio\5.0\Demos\Modeling\UML-2.0`.
3. Select `UML-2.0.tgproj` and click Open.
4. From the View menu, select **Model View**. Although the **Model View** opens initially as a free-floating window, it is a dockable window. The docking areas are any of the four borders of the RAD Studio window.
5. You can position the audit results window according to your preferences. In the **Model View**, expand the root project node and double-click the default diagram. The diagram opens in the **Diagram View**.

Double-click the various diagrams in the **Model View** to open them in the **Diagram View**.

See Also

UML 2.0 Sample Project (see page 84)

2.9.4 Together Diagram Procedures

This section provides how-to information on using Together UML diagrams.

Topics

Name	Description
Annotating a Diagram (see page 195)	
Creating a Diagram (see page 196)	When you create a new diagram, the **Diagram View** presents an empty background. You place the various model elements on the background and draw relationship links between them according to the requirements of your model.
Exporting a Diagram to an Image (see page 197)	
Printing a Diagram (see page 197)	You can print diagrams separately or as a group, or print all diagrams in the project.
Changing Diagram Notation (see page 197)	
Using Grid and Other Appearance Options (see page 198)	You can optionally display or hide a design grid on the diagram background and have elements “snap” to the nearest grid coordinate when you place or move them. The grid is configured in the Diagram Appearance options dialog window.
Using the UML in Color Profile (see page 198)	
Aligning Model Elements (see page 199)	You can automatically rearrange all or selected model elements on a diagram.
Changing Type of a Link (see page 199)	
Closing a Diagram (see page 200)	
Copying and Pasting Model Elements (see page 200)	The move and copy operations are performed by drag-and-drop, context menu commands, or keyboard shortcut keys. **Note:** You can move or copy an entire diagram. In this case, all elements addressed on this diagram are not copied, and a new diagram contains shortcuts to these elements.
Deleting a Diagram (see page 200)	**Warning:** The default diagram which is created automatically for a namespace (package) cannot be deleted.

Hyperlinking Diagrams (see page 201)	Select Hyperlinks from the diagram context menu to create, view, remove, and browse hyperlinks.
Laying Out a Diagram Automatically (see page 202)	
Moving Model Elements (see page 203)	Create your own layout by selecting and moving single or multiple diagram elements. You can: • Select a single element and drag it to a new position. • Select multiple elements and change their location. • Manually reroute links. **Note:** If you drag an element outside the borders of the Diagram View , the diagram automatically scrolls to follow the dragging. **Tip:** Manual layouts are saved when you close a diagram or project and restored when you next open it. Manual layouts are not preserved when you run one of the auto-layout commands (Do Full Layout or Optimize Sizes).
Renaming a Diagram (see page 204)	**Warning:** The project namespace (package) automatically created diagram cannot be renamed.
Rerouting a Link (see page 204)	
Resizing Model Elements (see page 204)	Diagram elements can be resized automatically or manually. When new items are added to an element that has never been manually resized, the element automatically grows to enclose the new items.
Selecting Model Elements (see page 205)	Most manipulations with diagram elements and links involve dragging the mouse or executing context menu commands on the selected elements.
Assigning an Element Stereotype (see page 205)	You can assign a stereotype in the diagram by using the in-place editor, or by using the Object Inspector.
Using Drag-and-Drop (see page 206)	Drag-and-drop applies to the members as well as to the node elements. You can move or copy members (methods, fields, properties, and so on) by using drag-and-drop in the **Diagram View** or in the **Model View**. Drag-and-drop functionality from the **Model View** to the **Diagram View** and within the **Model View** works as follows: • Selecting an element in the **Model View** and using drag-and-drop to place the element onto the diagram creates a shortcut. • Using drag-and-drop while pressing the `SHIFT` key moves the element to the selected container. • Using drag-and-drop while pressing the `CTRL` key copies the element to... more (see page 206)
Working with User Properties (see page 206)	User properties are created by means of the User Properties command. The User Properties command is available on the context menus of the diagrams and diagram elements both in the **Diagram View** and the **Model View**. Once created, the user properties can be viewed and edited in the Object Inspector under the User Properties category.
Creating a Link with Bending Points (see page 207)	If your diagram is densely populated, you can draw bent links between the source and target elements to avoid other elements that are in the way.
Creating Multiple Elements (see page 207)	You can place several elements of the same type on a diagram without returning to the Tool Palette or by using the diagram context menu. Each element will have a default name that can be edited with the in-place editor or in the Object Inspector.

Creating a Shortcut (↗ see page 208)	You can create a shortcut to a model element on the diagram background by using three methods: • By opening **Add Shortcuts** dialog box from the **Diagram View** • By copying and pasting a shortcut from the **Model View** • By choosing Add Shortcuts on the **Model View** context menu
Creating a Simple Link (↗ see page 209)	In a design project, you can create a link to another node, or a shortcut of an element of the same or another design project (these projects must be of the same UML version). In an implementation project, you can create a link to another node or a shortcut of an element of the same project.
Creating a Single Model Element (↗ see page 209)	
Searching Diagrams (↗ see page 209)	Together enables you to use the Find and Replace facilities provided by RAD Studio to locate model elements on model diagrams.
Searching Source Code for Usages (↗ see page 210)	In addition to the diagram search facility, Together enables you to track how an element or member is used in a source-code project. The **Search for Usages** dialog box enables you to find the references to, and overrides of, the elements and members in implementation projects. The Search for usages command is available on the context menu of an element in a diagram or in the **Model View**. Note that Search for usages is not available for the design projects.
Creating an Activity for a State (↗ see page 211)	
Designing a UML 1.5 Activity Diagram (↗ see page 211)	Use the following tips and techniques when you design a UML 1.5 Activity Diagram.
Instantiating a Classifier (↗ see page 211)	In a UML 1.5 design project, you can create an object that instantiates a class or interface from the same or another UML 1.5 design project or any implementation project in the same project group. In an implementation project, you can create an object that instantiates a class or interface from the same project or some UML 1.5 design project or a referenced project. You can create such links by using the Object Inspector or by using Dependency links to shortcuts.
Designing a UML 1.5 Component Diagram (↗ see page 212)	Following are tips and techniques that you can use when working with UML 1.5 Component Diagrams. It can be convenient to start creation of a model with Component Diagrams if you are modeling a large system. For example, a distributed, client-server software system, with numerous interconnected modules. You use Component Diagrams for modeling a logical structure of your system, while you use Deployment Diagrams for modeling a physical structure.
Designing a UML 1.5 Deployment Diagram (↗ see page 212)	Use the following tips and techniques when you design a UML 1.5 Deployment Diagram. It can be convenient to start creation of a model with Deployment Diagrams if you are modeling a large system that is comprised of multiple modules, especially if these modules reside on different computers. You use Deployment Diagrams for modeling a physical structure of your system, while you use Component Diagrams for modeling a logical structure.
Adding a Conditional Block (↗ see page 213)	**Note:** If the control structure requires a condition, you can enter the condition with the in-place editor, or you can enter it using the Condition field in the Object Inspector .
Associating an Object with a Classifier (↗ see page 214)	In the sequence or collaboration diagram you can create associations between objects (located on an interaction diagram) and classifiers (located on some class diagram). Instantiated classes for an object can be selected from the model, or the classes can be created and added to the model. Note that an object can instantiate classifiers that belong to the various source-code projects within a single project group, when such projects are referenced from the project in question. The range of available classifiers depends on the project type. • **Design projects**: classes, interfaces • **C# implementation projects**: classes, interfaces, structures
Branching Message Links (↗ see page 215)	Branching messages that start from the same location on the **lifeline**.
Converting Between UML 1.5 Sequence and Collaboration Diagrams (↗ see page 215)	You can convert between sequence and collaboration diagrams. However, when you create a new diagram, you must specify that it is either a sequence diagram or a collaboration diagram.

Working with a UML 1.5 Message (↗ see page 215)	This section describes techniques for working with messages in Sequence and Collaboration diagrams. Although the two diagram types are equivalent, the techniques for dealing with messages differ. In a Collaboration diagram, all messages between the two objects are displayed as a generic link line, and a list of messages is created above it. The link line is present as long as there is at least one message between the objects. Messages display in time-ordered sequence from top to bottom of the messages list. In addition to the message links, you can add links that show association and aggregation relationships. These... more (↗ see page 215)
Designing a UML 1.5 Statechart Diagram (↗ see page 217)	Following are tips and techniques that you can use when working with UML 1.5 Statechart Diagram.
Creating a Pin (↗ see page 217)	
Designing a UML 2.0 Activity Diagram (↗ see page 218)	Use the following tips and techniques when you design a UML 2.0 Activity Diagram. Usually you create Activity Diagrams after State Machine Diagrams.
Grouping Actions into an Activity (↗ see page 219)	
Working with an Object Flow or a Control Flow (↗ see page 219)	You can create control flow or object flow as an ordinary link between the two node elements. The valid nodes are highlighted when the link is established. You can scroll to the target element if it is out of direct reach, or you can use the context menu command to avoid scrolling. There are certain limitations stipulated by UML 2.0 specifications: • Object flow link must have an object at least on one of its ends. • It is impossible to connect two actions with an object flow except through an output pin on the source action. • Control flow link may not... more (↗ see page 219)
Designing a UML 2.0 Component Diagram (↗ see page 220)	Following are tips and techniques that you can use when working with UML 2.0 Component Diagrams. It can be convenient to start creation of a model with Component Diagrams if you are modeling a large system. For example, a distributed, client-server software system, with numerous interconnected modules. You use Component Diagrams for modeling a logical structure of your system, while you use Deployment Diagrams for modeling a physical structure.
Creating a Delegation Connector (↗ see page 221)	
Creating an Internal Structure for a Node (↗ see page 221)	
Creating a Referenced Part (↗ see page 221)	
Creating a Port (↗ see page 222)	
Working with a Collaboration Use (↗ see page 222)	
Designing a UML 2.0 Deployment Diagram (↗ see page 223)	Use the following tips and techniques when you design a UML 2.0 Deployment Diagram. It can be convenient to start creation of a model with Deployment Diagrams if you are modeling a large system that is comprised of multiple modules, especially if these modules reside on different computers. You use Deployment Diagrams for modeling a physical structure of your system, while you use Component Diagrams for modeling a logical structure.
Associating a Lifeline with a Classifier (↗ see page 224)	
Copying and Pasting an Execution or Invocation Specification (↗ see page 224)	Clipboard operations are supported for the execution and invocation specifications.
Creating a Sequence or Communication Diagram from an Interaction (↗ see page 225)	
Creating a State Invariant (↗ see page 225)	
Designing a UML 2.0 Sequence or Communication Diagram (↗ see page 226)	Use the following tips and techniques when you design a UML 2.0 Sequence or Communication Diagrams. Usually you create Interaction Diagrams after Class Diagrams. Whenever an interaction diagram is created, the corresponding interaction is added to the project. Interactions are represented as nodes in the **Model View**. **Note:** Presentation of an interaction in the Model View depends on the view type defined in the **Model View** options on the default or project group levels. If model-centric mode is selected, an interaction is shown both under its package node and diagram node. If diagram-centric mode is selected, an interaction is... more (↗ see page 226)
Linking Another Interaction from an Interaction Diagram (↗ see page 227)	
Working with a UML 2.0 Message (↗ see page 227)	This section describes techniques for working with messages in sequence and communication diagrams. Although the two diagram types are equivalent, the techniques for dealing with messages differ.
Working with a Combined Fragment (↗ see page 228)	

2

Working with a Tie Frame (see page 229)	
Associating a Transition or a State with an Activity (see page 230)	You can associate an activity (created on some UML 2.0 Activity Diagram) with a state (on entering the state, while doing the state activity, and on exiting the state), or with a transition between states.
Creating a Guard Condition for a Transition (see page 230)	
Creating a History Element (see page 230)	
Creating a Member for a State (see page 231)	
Creating a State (see page 231)	
Designing a UML 2.0 State Machine Diagram (see page 232)	Following are tips and techniques that you can use when working with UML 2.0 State Machine Diagram.
Browsing a Diagram with Overview Pane (see page 232)	
Hiding and Showing Model Elements (see page 232)	You can control the visibility of elements on a diagram by using the Hide command (available on the context menu for individual diagram elements), and the Show/Hide command (available on the diagram context menu).
Using View Filters (see page 233)	For global control over the diagram view, you can use the filters in the **Options** dialog window.
Zooming a Diagram (see page 234)	Use the diagram context menu to obtain the required magnification in the **Diagram View**.
Working with a Complex State (see page 234)	The techniques in this section pertain to models of particularly complex composite states and substates. You can resize the main state. You can also create a substate by drawing a state diagram within another state diagram and indicating start, end, and history states as well as transitions. Create a composite state by nesting one or more levels of states within one state. You can also place start/end states and a history state inside of a state, and draw transitions among the contained substates.
Creating a Deferred Event (see page 235)	You can add a deferred event to a state element.
Creating an Internal Transition (see page 235)	
Creating a Multiple Transition (see page 235)	
Creating a Self-Transition (see page 236)	
Specifying Entry and Exit Actions (see page 236)	You can create entry and exit actions as nodes, or as stereotyped **internal transitions**.
Working with an Instance Specification (see page 237)	You can instantiate a classifier using the Object InspectorProperties Window or the in-place editor.
Working with a Provided or Required Interface (see page 238)	
Creating an Association Class (see page 239)	
Creating an Inner Classifier (see page 239)	This section includes instructions for adding inner classifiers to classes (including Windows classes, such as Windows forms, Inherited forms, User Controls and so on), structures, and modules (collectively, containers) in implementation projects. You can add inner classifiers to class diagram elements (containers) using the respective context menu for the diagram element in the Diagram or **Model View**s. You can also select a classifier in the Tool PaletteToolbox and click the container element in the **Diagram View** to add the inner classifier to the container element. **Note:** Modules are specific to Visual Basic projects. Structure elements are available for... more (see page 239)
Using a Class Diagram as a View (see page 240)	Class diagrams can also be used to create subviews of the project.
Working with an Interface (see page 240)	This topic describes how to create and hide an interface on a class diagram.
Working with a Relationship (see page 241)	You can change the type of an association link.
Adding a Member to a Container (see page 241)	You can add members to class diagram elements (containers) by using the respective context menu for the diagram element in the **Diagram** or **Model Views** or available shortcut keys to add members to a class diagram container element.
Changing Appearance of Compartments (see page 242)	You can collapse or expand compartments for the different members of class, interface, namespace, module (Visual Basic projects only), enum, and structure (C# projects only) elements. By default, the compartments for these elements are displayed on the diagram as a straight line. You can use the **Options** dialog window to set viewing preferences for compartment controls. Adding compartment controls is particularly useful when you have large container elements with content that does not need to be visible at all times.
Changing Appearance of Interfaces (see page 242)	

Working with a Constructor (see page 243)	You can create as many constructors in a class as needed. In design projects, a constructor is created as an operation with the `<<constructor>>` stereotype. In implementation projects, each new constructor is created with its unique set of parameters. In addition to creating parameters automatically, you can define the custom set of parameters, using the Object InspectorProperties Window. **Tip:** You can move, copy and paste constructors and destructors between the container classes same way as the other members.
Working with a Field (see page 243)	This topic applies to implementation projects only. In the source code, it is possible to declare several fields in one line. This notation is represented in diagram as a number of separate entries in the Fields section if a class icon. However, you can rename the fields, change modifiers, set initial values and so on, all modifications being applied to the respective field in the diagram icon. Also you can copy and move such fields in diagram (using context menu commands or drag-and-drop), and the pasted field appears in the target container separately.
Associating a Message Link with a Method (see page 244)	Message links can be associated with the methods of the recipient class. The methods can be selected from the list of existing ones or can be created. This is done by two commands provided by the message context menu: Add and Choose method. You can use the **Operation** field in the Object InspectorProperties Window to rename the method. A dialog box appears asking if you want to create a new method or rename the old one.
Generating an Incremental Sequence Diagram (see page 245)	You can generate incremental sequence diagrams from a previously-generated sequence diagram. In some cases, you can have generated a sequence diagram with a low nesting value such as 3 or 5. The nesting value limits how deep the parser traverses the source code calling sequence.
Creating a Browse-Through Sequence of Diagrams (see page 246)	You can link entire diagrams at one level of detail to the next diagram up or down in a sequence of increasing granularity, or you can link from key use cases or actors to the next diagram.
Creating an Extension Point (see page 246)	
Designing Use Case Hierarchy (see page 246)	Use case diagrams typically represent the context of a system and system requirements.

2.9.4.1 Annotating a Diagram

Use the following actions to annotate a diagram:

1. Draw an annotation
2. Draw an annotation link
3. Type comments

To draw an annotation:

1. In the **Diagram View**, you can:

- Hyperlink the note to another diagram or element.
- Edit the text when its in-place editor is active.
- Edit the properties of a note using Object Inspector.
- Add an existing note from one diagram to another diagram using a shortcut. (Select **Add** ▶**Shortcuts** from any diagram context menu.)

2. In the Object Inspector for the note, you can:

- Edit the text.
- Change the foreground and background colors.
- Change the text-only property.

To draw an annotation link:

1. Click the **Note Link** button on the Tool Palette.

2. In the **Diagram View**, click the source element.
3. Drag the link to the destination element.
4. Drop when the second element is highlighted.

   Tip: You can use the Object Inspector

   to view both the client and supplier sides of the link.

To type comments:

1. To enter comments in the source code, use the **Comment** fields (Author, Since, Version) in the Object Inspector for the class.
2. You can also enter source code comments directly into the code using the Editor.

See Also

Annotation Overview (see page 91)

Creating a Single Element (see page 209)

Creating a Shortcut (see page 208)

2.9.4.2 Creating a Diagram

When you create a new diagram, the **Diagram View** presents an empty background. You place the various model elements on the background and draw relationship links between them according to the requirements of your model.

To create a diagram:

1. In the **Model View**, right-click the target project.

   Tip: Alternatively, you can use the shortcut CTRL+SHIFT+D

   .
2. Select the target namespace (package) either in the **Diagram View** or in the **Model View**. If you do not select a custom namespace (package), Together adds a new diagram to the default one.
3. Choose **Add ▶ Other Diagram** on the context menu.
4. In the **Add New Diagram** dialog box, choose the **Diagrams** tab.
5. Select the diagram type.
6. In the **Name** field, enter a name for the new diagram.
7. Click **OK**.

Result: The new diagram opens in a new tab in the **Editor Window**. You can use the Object Inspector to view and edit the diagram properties.

To create a new diagram, use can also use the **Hyperlink ▶ To New diagram** command on the context menu of the **Model View** or the **Diagram View**.

You can create a new logical class diagram using the context menu of the root node for your project, or by using the context menu of a namespace element in the **Model View**. Choose either **Add ▶ Class Diagram** or **Add ▶ Other Diagram**. Choosing the latter command opens the **Add New Diagram** dialog box. When you place a class, interface, or namespace on a logical class diagram, Together generates the corresponding source code or descendent namespace in the namespace where this class diagram is located.

See Also

Diagram Overview (see page 90)

Creating a Project (see page 264)

2.9.4.3 Exporting a Diagram to an Image

To export a diagram to an image:

1. Place the focus on the diagram you want export in the **Diagram View**.
2. Choose **File ▶ Export Diagram to Image** on the main menu. The **Export Diagram to Image** dialog opens.
3. Click the drop-down arrow to preview and adjust the zoom settings of the diagram image.
4. Click **Save**. The file browser dialog box opens.
5. Browse for a location where you wish to save the image.
6. Enter a name. By default, the image file takes on the name given to the diagram in RAD Studio.
7. Select an image format.
8. Click **Save**.

See Also

Import and Export Features Overview (↗ see page 100)

2.9.4.4 Printing a Diagram

You can print diagrams separately or as a group, or print all diagrams in the project.

To print a diagram:

1. With the diagram in focus in the **Diagram View**, choose **File ▶ Print** from the main menu. The **Print diagram** dialog box opens.
2. In the Print Diagrams list box, specify the scope of diagrams to be printed:

- **Active diagram**: To print the currently selected diagram.
- **Active with neighbors**: To print the current diagram and the other diagrams of the same project.
- **All opened**: To print all diagrams currently opened in the Diagram view.
- **All in model**: To print all diagrams within a project group.

3. In the Print zoom field, specify the zoom factor.
4. If necessary, adjust the page and printer settings:

- Click the Print list box and choose Print dialog box to select the target printer.
- Use the Options dialog window (**Together ▶ (level) ▶ Diagram ▶ Print** options) to set up the paper size, orientation, and margins.

Tip: Click Preview to open the preview pane. Use the Preview zoom slider, or Auto Preview zoom check box, as required.

See Also

Diagram Print options (↗ see page 1094)

2.9.4.5 Changing Diagram Notation

Use the following techniques to change diagram notation:

1. Choose one of the two possible appearances for interfaces. Interfaces can be represented as rectangles or small circles ("lollipops").

2. In UML 2.0 projects, you can change notation of interfaces to "ball and socket".
3. Adjust appearance options, including selection between UML or language formats.

   Tip: Notation options are included in the Diagram ->Appearance

   category of Together options.
4. Use the **UML In Color** profile.
5. Use stereotypes.

See Also

Diagram Layout Overview (see page 92)

Changing Appearance of Interfaces (see page 242)

Using the “UML In Color” Profile (see page 198)

Diagram Appearance Notation Options (see page 1089)

2.9.4.6 Using Grid and Other Appearance Options

You can optionally display or hide a design grid on the diagram background and have elements “snap” to the nearest grid coordinate when you place or move them. The grid is configured in the Diagram Appearance options dialog window.

To show grid:

1. Open Options dialog window.
2. Choose the **Together**▶**Diagram**▶**Appearance** category, Grid group.
3. Adjust the options.

   Note: Grid display and snap are enabled by default.

See Also

Diagram Appearance options (see page 1089)

2.9.4.7 Using the UML in Color Profile

To enable or disable the “UML in color” profile:

1. In the **Options** dialog window, open the **Together**▶**(level)**▶**Diagram**▶**Appearance** category.

   Tip: You can enable or disable it on for the project group

   , project, or diagram level.
2. Set the **Enable UML in color** option to *True* to enable the profile.
3. Optionally, adjust colors used by the profile.
4. Close the **Options** dialog window.

To draw UML nodes in colors:

1. Select or create a classifier.
2. Open the Object Inspector.
3. Assign a stereotype that is supported by the “UML in color” profile (for example, *role*).

Result: The classifier changes its color according to the settings in the **Options** dialog window.

See Also

Supported UML Specifications (see page 90)

2.9.4.8 Aligning Model Elements

You can automatically rearrange all or selected model elements on a diagram.

To align model elements on a diagram:

1. Select several nodes or inner classifiers on a diagram.
2. Right-click and choose **Alignment ▶(algorithm)** on the context menu. The following algorithms are available:

- Top
- Bottom
- Rigth
- Left
- Center X
- Center Y

See Also

Laying out a diagram automatically (see page 202)

2.9.4.9 Changing Type of a Link

Use the following techniques to change the type of a link:

1. Set the link type by using the Object Inspector
2. Set the link type by using the context menu

To set the link type by using the name="Delphi"Object Inspector:

1. Choose View | Object Inspector if the Object Inspector is not open.
2. Select a link on the diagram. The properties for the link appear in the Object Inspector.
3. In the Object Inspector, select the **Type** field.
4. Click the drop-down arrow and select the appropriate property from the list. Your available choices are association, aggregation, or composition.

To set the link type by using the context menu:

1. Right-click a link on the diagram.
2. Choose Link Type on the context menu.

See Also

Creating a Simple Link (see page 209)

Class Diagram Relationships (see page 1123)

2.9.4.10 Closing a Diagram

To close a diagram:

1. Switch to the **Diagram View**.
2. Click the cross icon to close the current view.

   Note: Closing a diagram in the Diagram View

   does not remove it from your project.

See Also

Diagram Overview (see page 90)

2.9.4.11 Copying and Pasting Model Elements

The move and copy operations are performed by drag-and-drop, context menu commands, or keyboard shortcut keys.

Note: You can move or copy an entire diagram. In this case, all elements addressed on this diagram are not copied, and a new diagram contains shortcuts to these elements.

To copy an element:

1. Select the element or elements to be copied.
2. Do any of the following:

- Right-click and choose Copy on the context menu
- Press `CTRL+C` on the keyboard

3. Do any of the following:

- Right-click the target location and choose Paste on the context menu
- Select the target location and press `CTRL+V`

See Also

Creating a single element (see page 209)

Keyboard shortcuts (see page 1104)

2.9.4.12 Deleting a Diagram

Warning: The default diagram which is created automatically for a namespace (package) cannot be deleted.

To delete a diagram:

1. In the **Model View**, select the diagram to be deleted.
2. On the context menu, choose Delete.
3. Confirm deletion, if required.

Result: The diagram is deleted from the project.

See Also

Creating a Diagram (see page 196)

Closing a Diagram (↗ see page 200)

2.9.4.13 Hyperlinking Diagrams

Select Hyperlinks from the diagram context menu to create, view, remove, and browse hyperlinks.

Use the following techniques to create a hyperlink:

1. Create a hyperlink to an existing diagram or element
2. ?reate a hyperlink to a new diagram
3. ?reate a hyperlink to an external URL or file
4. Browse hyperlinks
5. Remove a hyperlink

To create a hyperlink to an existing diagram or element:

1. Open an existing diagram or create a new diagram from which to create the hyperlink.
2. Select the element that you want to link to another diagram or element.
3. To link the entire diagram, click the diagram background to deselect all elements.

   Note: Do not select the actual namespace in the Model View

   to create a hyperlink. Rather, expand the namespace node, and select the desired diagram.
4. Right-click and choose **Hyperlinks ▶ Edit**. The **Edit Hyperlinks** dialog window (Selection Manager) opens.
5. Select the Model Elements tab to view the pane containing a tree view of the available project contents in the Solution.
6. Select the desired diagram or element from the list, and click **Add**.
7. For element selection, expand diagram nodes in the **Model Elements** tab.
8. To remove an element from the selected list, select the element and click **Remove**.
9. Click **OK** to close the dialog box and create the link.

To create a hyperlink to a new diagram:

1. Open a diagram in the **Diagram View**, or select it in the **Model View**.
2. On the context menu, choose **Hyperlinks ▶ To New Diagram**.
3. In the **Add New Diagram** dialog box, select the diagram type, enter the diagram name and click OK.

To create a hyperlink to an external URL or file:

1. Open an existing diagram or create a new diagram from which to create the hyperlink.
2. Select the element that you wish to link to the external document. To link the entire diagram, click the diagram background to deselect all elements.
3. Right-click and choose **Hyperlinks ▶ Edit**. The **Edit Hyperlinks** dialog box opens.
4. Select the **External Documents** tab to view the Recently used Documents list which contains a list of previously selected files or URLs.
5. To add a file to the Recently used Documents list:
1. Click **Browse**. The **Open file** dialog box opens.
2. Navigate to the desired file and click **Open**.
6. To add a URL to the Recently used Documents list:

3. Click URL.
4. In the dialog box that opens, enter the appropriate URL and click **OK**.

   Tip: You can create a hyperlink to an external document by entering a relative URL path.

7. To remove an element from the selected list, select the element and click **Remove**.
8. To clear the Recently used Documents list, click **Clear**.

   Note: Items added to the Recently used Documents list are not specific to a single project or project group

   .

9. Click **OK** to close the dialog box and create the link.

To browse hyperlinks:

1. To view hyperlinks to a diagram, element or external document, right-click on the diagram background or element, and choose Hyperlinks from the context menu. All hyperlinks created appear under the Hyperlinks submenu. On a diagram, all names of diagram elements that are hyperlinked are displayed in blue font. When you select a link from the submenu, the respective element appears selected in the **Diagram View**.
2. Once you have defined hyperlinks for a selected diagram or element, use the context menus to browse to the linked resources.

   Note: Browsing to a linked diagram opens it in the Diagram View

   or makes it the current diagram if already open. Browsing to a linked element causes its parent diagram to open or become current, and the diagram scrolls to the linked element and selects it.

To remove a hyperlink:

2

1. Open the diagram that displays the link you want to remove.
2. Choose **Hyperlinks ▸ Edit** from the diagram or element context menu. The **Edit Hyperlinks** dialog box opens.
3. In the selected list on the right of the dialog, click the hyperlink that you wish to remove.
4. Click **Remove**.
5. Click **OK** to close the dialog box.

   Note: To remove a hyperlink from a specific element, select the element first. Then choose Hyperlinks->Edit

   on the context menu.

See Also

Hyperlinking Overview (see page 92)

2.9.4.14 Laying Out a Diagram Automatically

To lay out a diagram by using one of the algorithms:

1. Right-click the diagram background.
2. On the context menu, select Layout, and choose a command from the submenu. There are several Layout commands on the Layout submenu:

- **Do Full Layout**: Sets the layout of all elements according to the layout algorithm defined for the current diagram.
- **Layout for Printing**:Sets the layout of all elements using the *Together* algorithm, regardless of the option selected on any level.
- **Route All Links**:Streamlines the links removing bending points.
- **Optimize Sizes**: Enlarges or shrinks all elements on the diagram to the optimal size.

Note: Individual diagram elements also have the Route Links and Optimize Size layout commands on their respective context menus. The Route Links command streamlines the links removing any bending points. The Optimize Size command enlarges or shrinks the element to the optimal size, leaving enough space for its label and any sub elements it may contain.

Tip: To enable layout of the inner substructure in diagrams, check the Recursive

option (**(level)** ▶ **Diagram** ▶ **Layout** ▶ **General**) in the **Options** dialog window.

To set up the diagram layout:

1. On the main menu choose **Tools** ▶ **Options**.
2. On the desired level, select **Together** ▶ **(level)** ▶ **Diagram** ▶ **Layout** category.
3. Expand the node for the desired algorithm.
4. Specify the algorithm-specific options (if any) and apply changes.

Result: you can observe results of layout tuning when apply one of the *Layout* commands to the diagram.

The context menu available in the **Diagram View** provides access to the automated layout optimization features in Together.

See Also

Diagram Layout Overview (↗ see page 92)

Aligning Model Elements (↗ see page 199)

Layout Diagram Options (↗ see page 1091)

2.9.4.15 Moving Model Elements

Create your own layout by selecting and moving single or multiple diagram elements.

You can:

- Select a single element and drag it to a new position.
- Select multiple elements and change their location.
- Manually reroute links.

Note: If you drag an element outside the borders of the Diagram View

, the diagram automatically scrolls to follow the dragging.

Tip: Manual layouts are saved when you close a diagram or project and restored when you next open it. Manual layouts are not preserved when you run one of the auto-layout commands (Do Full Layout or Optimize Sizes).

To move an element:

1. Select the element or elements to be moved.
2. Drag-and-drop the selection to the target location.

Tip: Right-click and use Cut and Paste. Use the keyboard shortcuts for Cut (CTRL+X

), Copy (`CTRL+C`), and Paste (`CTRL+V`) operations.

See Also

Select Model Elements (↗ see page 205)

Keyboard shortcuts (↗ see page 1104)

2.9.4.16 Renaming a Diagram

Warning: The project namespace (package) automatically created diagram cannot be renamed.

To rename a diagram:

1. In the Object Inspector, double-click the diagram name to initiate the inline editor.
2. Enter a new name.
3. Press Enter.

Alternatively:

1. Select the diagram in the **Model View**.
2. Press `F2` or right-click and choose Rename on the context menu.
3. Enter a new name.
4. Press `Enter`.

Result: The diagram is renamed.

See Also

Creating a diagram (see page 196)

2.9.4.17 Rerouting a Link

To reroute a link:

1. Select a link.
2. Drag and drop the client of supplier end of the link to the desired destination object.
3. To change direction of the link, click a place on the link where you want to reroute the link.
4. Drag the line. Together automatically reshapes the link the way you want.

   Tip: Model elements have the Layout->Route All Links

   command on diagram context menus.

See Also

Model Element Overview (see page 91)

Laying Out a Diagram (see page 202)

2.9.4.18 Resizing Model Elements

Diagram elements can be resized automatically or manually. When new items are added to an element that has never been manually resized, the element automatically grows to enclose the new items.

To resize an element manually:

1. Click an element. The selected element is highlighted with bullets.
2. Drag one of the bullets in the desired direction.

When the element contents change, for example, when members are added or deleted, and the element size is too small to display all members, scroll bars are displayed to the right of compartments.

To optimize a node element size:

1. Right-click an element.
2. Choose **Layout** ▶ **Optimize Size**.

To optimize the elements on an entire diagram:

1. Right-click the diagram background.
2. Choose **Layout** ▶ **Optimize Size**.

See Also

Lay Out a Diagram Automatically (see page 202)

2.9.4.19 Selecting Model Elements

Most manipulations with diagram elements and links involve dragging the mouse or executing context menu commands on the selected elements.

To select a model element:

1. Open the **Diagram View**.
2. On a diagram:

- Click any element in the diagram to select it.
- To select multiple elements, hold down the `CTRL` key and click each element individually.
- Click the background and drag a lasso around an area to select all the elements it contains.
- For elements containing members, click on a member to select it.
- To select all elements on a diagram, press `CTRL+A`. Alternatively, right-click the diagram background and choose Select All on the context menu.

See Also

Aligning model elements (see page 199)

Keyboard shortcuts (see page 1104)

2.9.4.20 Assigning an Element Stereotype

You can assign a stereotype in the diagram by using the in-place editor, or by using the Object Inspector.

Use the following techniques to specify a stereotype:

1. Assign a stereotype by using the in-place editor
2. Assign a stereotype by using the Object Inspector

To assign a stereotype by using the in-place editor:

1. Double-click the stereotype name to activate the in-place editor.
2. Enter the new name.
3. Press `Enter`.

To assign a stereotype by using the name="Delphi"Object Inspector:

1. Select a class on your diagram.
2. In the Object Inspector, select the **Stereotype** field.
3. Click the value editor button and choose the required stereotype from the combo box. Alternatively, type the stereotype name.

Result: The stereotype name is displayed in angle brackets in the class node.

See Also

Using UML in color (see page 198)

2.9.4.21 Using Drag-and-Drop

Drag-and-drop applies to the members as well as to the node elements. You can move or copy members (methods, fields, properties, and so on) by using drag-and-drop in the **Diagram View** or in the **Model View**.

Drag-and-drop functionality from the **Model View** to the **Diagram View** and within the **Model View** works as follows:

- Selecting an element in the **Model View** and using drag-and-drop to place the element onto the diagram creates a shortcut.
- Using drag-and-drop while pressing the `SHIFT` key moves the element to the selected container.
- Using drag-and-drop while pressing the `CTRL` key copies the element to the selected container.

Tip: You can also change the origin and destination for links on your diagrams using drag-and-drop.

To move a link to a new destination:

1. Select a link in the **Diagram View**.
2. Hover the cursor over the destination arrow.
3. Drag the arrow and drop it on the new destination. If the destination element is not in view, drag the link in the appropriate direction, and the diagram will scroll with you.

Tip: Follow the same instructions to move the link source to an allowable location.

See Also

Selecting a Model Element (see page 205)

Moving a Model Element (see page 203)

Keyboard Shortcuts (see page 1104)

2.9.4.22 Working with User Properties

User properties are created by means of the User Properties command. The User Properties command is available on the context menus of the diagrams and diagram elements both in the **Diagram View** and the **Model View**. Once created, the user properties can be viewed and edited in the Object Inspector under the User Properties category.

To create user properties:

1. In the **Diagram View** or the **Model View**, select the desired diagram or model element.
2. On the context menu, choose User Properties.
3. In the **Add/Remove user properties** dialog box, click the Add button. A new entry, consisting of the Name and Value fields, is added to the properties list.
4. In the new entry, enter the property name and value.

5. Using the Add and Remove buttons, make up the list of user properties.
6. Click **OK** when ready.

Result: The User Properties category appears in the Object Inspector.

2.9.4.23 Creating a Link with Bending Points

If your diagram is densely populated, you can draw bent links between the source and target elements to avoid other elements that are in the way.

To create a link with bending points:

1. Click the link button on the Tool Palette.
2. Click the source element.
3. Drag the link line, clicking the diagram background each time you want to create a section of the link. Sections on a link lie between two blue bullets. The bullets display whenever you select the link on the diagram.
4. Click the destination element to terminate the link.

   Tip: Once you have created a link, you can add bending points to it. Select the link on the diagram, and then drag the link to the desired position.

See Also

Rerouting a Link (↗ see page 204)

Creating a Simple Link (↗ see page 209)

Class Diagram Relationships (↗ see page 1123)

2.9.4.24 Creating Multiple Elements

You can place several elements of the same type on a diagram without returning to the Tool Palette or by using the diagram context menu. Each element will have a default name that can be edited with the in-place editor or in the Object Inspector.

To create multiple elements:

1. Holding down the `CTRL` key, click the Tool Palette button for the element you want to create (the button stays down). Release the `CTRL` key.
2. Click the desired location on the diagram background. The new element is placed on the diagram at the point where you click.
3. Click the next location on the diagram background. The next new element is placed on the diagram.
4. Repeat the previous step until you have the desired number of elements of that type.
5. To stop multiple element creation, click the Pointer Tool Palette button or press the `ESC` key to deselect the element after closing the in-place editor of the last inserted element.

   Tip: After making a selection on the Tool Palette

   or doing the first of a multi-draw or multi-placement operation, you can cancel the operation by clicking the Pointer button on the Tool Palette or by pressing the `ESC` key.

See Also

Creating a single element (↗ see page 209)

Creating a simple link (↗ see page 209)

Keyboard shortcuts (↗ see page 1104)

2.9.4.25 Creating a Shortcut

You can create a shortcut to a model element on the diagram background by using three methods:

- By opening **Add Shortcuts** dialog box from the **Diagram View**
- By copying and pasting a shortcut from the **Model View**
- By choosing Add Shortcuts on the **Model View** context menu

Use the following techniques to create a shortcut:

1. Create a shortcut by using the **Add Shortcuts** dialog window
2. Create a shortcut by using drag-and-drop
3. Create a shortcut by copying and pasting
4. Create a shortcut by using the **Model View** context menu

To create a shortcut by using the Add Shortcuts dialog window:

1. Right-click the diagram background.
2. Choose **Add ▶ Shortcuts** on the context menu.

   Tip: You can also use CTRL+SHIFT+M

   to open the **Edit shortcuts** dialog window.
3. In the **Edit shortcuts** dialog window, choose the required element from the tree view of available contents.
4. Click **Add** to place the selected element to the list of the existing or ready to add elements.
5. When the list of ready to add elements is complete, click **OK**.

To create a shortcut by using drag-and-drop:

1. Select the element in the **Model View**.
2. Drag-and-drop the element onto the diagram.

To create a shortcut by copying and pasting:

1. In the **Model View**, right-click the element to be added to the current diagram as a reference.
2. Choose Copy on the context menu.
3. Right-click the target diagram and choose Paste Shortcut on the context menu.

   Tip: You can also copy an element from one diagram and paste it in another diagram as a shortcut.

To create a shortcut by using the Model View context menu:

1. Open the diagram where the shortcut will be added.
2. In the **Model View**, select the element to be added to the current diagram as a shortcut.
3. Right-click the element in the **Model View**, and choose Add as Shortcut on the context menu.

See Also

Shortcut Overview (see page 92)

Hyperlinking Overview (see page 92)

Creating a Single Element (see page 209)

2.9.4.26 Creating a Simple Link

In a design project, you can create a link to another node, or a shortcut of an element of the same or another design project (these projects must be of the same UML version).

In an implementation project, you can create a link to another node or a shortcut of an element of the same project.

To create a simple link between two nodes:

1. On the diagram Tool Palette, click the button for the type of link you want to draw in the diagram. The button stays down.
2. Click the source element.
3. Drag to the destination element and drop when the second element is highlighted.

See Also

Rerouting a link (see page 204)

Creating a link with bending points (see page 207)

Creating a link by pattern (see page 255)

Class diagram relationships (see page 1123)

2.9.4.27 Creating a Single Model Element

To create a single model element:

1. Open a target diagram in the **Diagram View**.
2. Choose Tool Palette from the View menu.
3. Choose the **UML [diagram type]** tab in the Tool Palette to view available model elements.
4. On the Tool Palette, click the icon for the element you want to place on the diagram. The button stays down.

   Tip: Icons are identified with labels.

5. Click the diagram background in the place where you want to create the new element. This creates the new element and activates the in-place editor for its name.

   Tip: Alternatively, you can right-click the diagram background and choose Add on the context menu. The submenu displays all of the basic elements that can be added to the diagram, and the Shortcuts command.

See Also

Creating Multiple Model Elements (see page 207)

Creating a Simple Link (see page 209)

2.9.4.28 Searching Diagrams

Together enables you to use the Find and Replace facilities provided by RAD Studio to locate model elements on model diagrams.

To search diagrams:

1. Choose Search->(search command) to use the find and replace facilities provided by the RAD Studio.

2. You can find the specified string in the specified scope. The function supports case sensitivity, searching for whole words or substrings, using wildcards and regular expressions.
3. Browse the search results.

See Also

Searching source code for usages (see page 210)

2.9.4.29 Searching Source Code for Usages

In addition to the diagram search facility, Together enables you to track how an element or member is used in a source-code project. The **Search for Usages** dialog box enables you to find the references to, and overrides of, the elements and members in implementation projects.

The Search for usages command is available on the context menu of an element in a diagram or in the **Model View**. Note that Search for usages is not available for the design projects.

To search source code for element usages:

1. Right-click an element or a namespace and choose Search for Usages on the context menu. The dialog box opens with the selected element specified in the section Element to search.
2. In the Options section, check the following options as required:

- Usages of elements
- Usages of members
- Usages of Declared Classes
- Implementations
- Overriding
- Include usings/imports
- Skip self

3. Click Search.

The search results are displayed in a tab in the Search for Usages window as a tree view, each node containing all usages of an element in a certain class. Note that each new search adds its own tab to the window.

The Search for Usages window provides a toolbar with the buttons that enable you to expand or collapse the treeview nodes, and repeat the search in the selected tab with the same settings.

The context menu of a search results tab provides the following commands:

Command
Close
Close all
Close all but this

See Also

Searching Diagrams (see page 209)

2.9.4.30 Creating an Activity for a State

To create an activity for a state:

1. Open the **Diagram View**.
2. Right-click a state and choose **Add** ▶ **Activity** on the context menu.

Result: A new activity is created inside of a state.

See Also

UML 1.5 Activity Diagram (↗ see page 1117)

2.9.4.31 Designing a UML 1.5 Activity Diagram

Use the following tips and techniques when you design a UML 1.5 Activity Diagram.

To design a UML 1.5 Activity Diagram, follow this general procedure:

1. Create one or more swimlanes. You can place several swimlanes on a single diagram, or create a separate diagram for each.

   Warning: You cannot create nested swimlanes.

2. Create one or more activities. You can place several activities on a single swimlane, or create a separate swimlane for each.

   Warning: You cannot create nested activities.

3. For convenient browsing, first model the main flow. Next, cover branching, concurrent flows, and object flows.

   Tip: Use separate diagrams as needed and then hyperlink them.

4. Create **Start**, **End**, **Signal Receipt**, and **Signal Sending** elements for your swimlanes. If your activity has several **Start** points, they can be used simultaneously.
5. Create object nodes. You do not link object nodes to classes on your Class Diagrams. However, you can use hyperlinks for better understanding of your diagrams.
6. Create state nodes for your swimlanes.

   Tip: You can create nested states.

7. Optionally, create a **History** node.
8. Connect nodes by links.
9. You can optionally create shortcuts to related elements of other diagrams.

See Also

Creating a Shortcut (↗ see page 208)

UML 1.5 Activity Diagram Reference (↗ see page 1117)

2.9.4.32 Instantiating a Classifier

In a UML 1.5 design project, you can create an object that instantiates a class or interface from the same or another UML 1.5

design project or any implementation project in the same project group. In an implementation project, you can create an object that instantiates a class or interface from the same project or some UML 1.5 design project or a referenced project. You can create such links by using the Object Inspector or by using Dependency links to shortcuts.

To instantiate a classifier:

1. On a UML 1.5 class diagram, choose an object.
2. In the Object Inspector, choose the **Instantiates** field.
3. Click the **Chooser** button. The **Choose Type to Instantiate** dialog box opens.
4. In this dialog box, choose a classifier (class or interface).

   Tip: Alternatively, draw a Dependency link

   from this object to a classifier or its shortcut.

See Also

UML 1.5 Class diagram (↗ see page 1121)

2.9.4.33 Designing a UML 1.5 Component Diagram

Following are tips and techniques that you can use when working with UML 1.5 Component Diagrams. It can be convenient to start creation of a model with Component Diagrams if you are modeling a large system. For example, a distributed, client-server software system, with numerous interconnected modules. You use Component Diagrams for modeling a logical structure of your system, while you use Deployment Diagrams for modeling a physical structure.

To design a UML 1.5 Component Diagram, follow this general procedure:

1. Create a hierarchy of Subsystems.

   Tip: You can create nested Subsystems.

2. Create a hierarchy of Components. The largest component can be the whole system or its major part (for example, *server application*, *IDE*, *service*).

   Tip: You can create nested component nodes. There are two methods for creating a nested component node: You can select an existing component and add a child component inside. Alternatively, you can create two separate components and connect them with an Association-Composition link.

3. Create interfaces. Each component can have an interface.
4. Draw links between elements.
5. You can optionally create shortcuts to related elements of other diagrams.

See Also

Creating a Shortcut (↗ see page 208)

UML 1.5 Component Diagram Reference (↗ see page 1128)

2.9.4.34 Designing a UML 1.5 Deployment Diagram

Use the following tips and techniques when you design a UML 1.5 Deployment Diagram. It can be convenient to start creation of a model with Deployment Diagrams if you are modeling a large system that is comprised of multiple modules, especially if these modules reside on different computers. You use Deployment Diagrams for modeling a physical structure of your system, while you use Component Diagrams for modeling a logical structure.

To design a UML 1.5 Deployment Diagram, follow this general procedure:

1. Create a hierarchy of Nodes.

   Tip: You can create nested Nodes.

2. Create a hierarchy of Components. The largest component can be the whole system or its major part (for example, *server application*, *IDE*, *service*).

   Tip: You can create nested Components. There are two methods for creating a nested component: You can select an existing component and add a child component inside. Alternatively, you can create two separate components and connect them with an Association-Composition link.

3. Represent how Components resides on Nodes. You can represent this in two ways:

- Use a supports link between the component and node. The supports link is a dependency link with the stereotype field set to support.
- Graphically nest the Component within the Node.

4. Optionally, create Objects.
5. Create Interfaces. Each component can have an interface.
6. Indicate a temporary relationship between a Component and Node. Objects and components can migrate from one component instance to another component instance, and respectively from one node instance to another node instance. In such a case, the object (component) will be on its component (node) only temporarily. To indicate this, use the dependency relationship with a becomes stereotype.
7. You can optionally create shortcuts to related elements of other diagrams.

See Also

Creating a Shortcut (see page 208)

UML 1.5 Deployment Diagram Reference (see page 1129)

2.9.4.35 Adding a Conditional Block

Note: If the control structure requires a condition, you can enter the condition with the in-place editor, or you can enter it using the Condition field in the Object Inspector

.

To add a statement block to the activation bar:

1. In the Tool Palette, click the Conditional Block button.
2. Click the target activation bar.

Alternatively:

1. Right-click an activation bar on a sequence diagram.
2. Choose **Add** ▶**Conditional Block** on the context menu.

To set the type of the conditional block (if, for, and so on):

1. Open the Object Inspector.
2. Click the drop-down arrow for your choices.

See Also

UML 1.5 interaction diagrams (↗ see page 1131)

2.9.4.36 Associating an Object with a Classifier

In the sequence or collaboration diagram you can create associations between objects (located on an interaction diagram) and classifiers (located on some class diagram). Instantiated classes for an object can be selected from the model, or the classes can be created and added to the model.

Note that an object can instantiate classifiers that belong to the various source-code projects within a single project group, when such projects are referenced from the project in question.

The range of available classifiers depends on the project type.

- **Design projects**: classes, interfaces
- **C# implementation projects**: classes, interfaces, structures

To associate an object with an existing classifier:

1. Select an object.
2. On the context menu of the object, select Choose class.
3. The submenu displays the list of available classifiers. If you cannot find the required classifier in the list, click More to reveal the model tree view.
4. In the **Choose Type to Instantiate** dialog box that opens, select a classifier from the model and click **OK**.

   Tip: Alternatively, use the Object Inspector

   . Click the Instantiates field and select the classifier from the model. Result: The object displays the fully qualified path to the instantiated classifier.

   Tip: To associate an object with a classifier from a different project, add this project as a referenced one.

To create a new classifier for an existing object:

1. Select an object.
2. On the context menu, choose Add.
3. From the submenu, choose the desired classifier type.

Result: A new classifier is added to the model. A shortcut for the new classifier appears on the interaction diagram in question, connected with the object by a dependency link.

To unlink an object:

1. Select an object.
2. On the context menu of the object choose Unlink class.

Result: The association is removed, but the classifier is preserved in the model.

To navigate between classifiers and objects:

1. Select the object on the diagram.
2. Right-click and choose Synchronize Model View on the context menu to move focus to this classifier in the **Model View**, or choose Go to Class Definition to open this classifier in the source code (for implementation projects).

To create a shortcut to a classifier on an interaction diagram:

1. On the diagram, select an object that instantiates a classifier.

2. Right-click and choose Import class on the context menu.

Result: A shortcut to the instantiated classifier is added to the diagram.

See Also

Working with a referenced project (see page 270)

UML 1.5 Class diagrams (see page 1121)

UML 1.5 Interaction diagrams (see page 1131)

2.9.4.37 Branching Message Links

Branching messages that start from the same location on the **lifeline**.

To branch a message link with the previous one:

1. Select a message link on the sequence or collaboration diagram.
2. Right-click the message link and choose **Branching ▶ With previous** on the context menu.

To remove branching:

1. Select the message link to remove branching from.
2. Right-click the message link and choose **Branching ▶ None** on the context menu.

See Also

Working with UML 1.5 messages (see page 215)

UML 1.5 interaction diagrams (see page 1131)

2.9.4.38 Converting Between UML 1.5 Sequence and Collaboration Diagrams

You can convert between sequence and collaboration diagrams. However, when you create a new diagram, you must specify that it is either a sequence diagram or a collaboration diagram.

To convert between sequence and collaboration diagrams:

1. Right-click the diagram background.
2. If the diagram is a sequence diagram, choose Show as Collaboration on the context menu. If the diagram is a collaboration diagram, choose Show as Sequence.
3. Repeat this process to switch back and forth.

After you convert from a sequence diagram to a collaboration diagram for the first time, or if you have added new objects to the sequence diagram between conversions, it is recommended that you perform a full layout on the collaboration diagram.

See Also

UML 1.5 interaction diagram (see page 1131)

2.9.4.39 Working with a UML 1.5 Message

This section describes techniques for working with messages in Sequence and Collaboration diagrams. Although the two diagram types are equivalent, the techniques for dealing with messages differ.

In a Collaboration diagram, all messages between the two objects are displayed as a generic link line, and a list of messages is created above it. The link line is present as long as there is at least one message between the objects. Messages display in time-ordered sequence from top to bottom of the messages list. In addition to the message links, you can add links that show association and aggregation relationships. These links do not display if you view the diagram as a sequence diagram.

When you draw messages between objects in a sequence diagram, each message is represented by its own link line. Messages in sequence diagrams have more editable properties than messages in collaboration diagrams.

Use the following techniques for messages:

1. ?reate a self message
2. Reorder a message link
3. Specify creation of an object with a message
4. Specify destruction of an object with a message
5. Specifying a return link by using the Tool Palette (Toolbox)
6. Specify a return link by using the Object Inspector (Properties Window)

To create a self message:

1. Click the **Self Message** button on the Tool Palette.
2. For a Sequence diagram, click the lifeline of the object at the point where you want the message to appear. Clicking the object places the message-to-self first on the lifeline. For a Collaboration diagram, click the object.

To reorder a message link:

1. Open a diagram.
2. To reorder messages, perform one of the following actions:

- Drag message links up and down the object lifeline in the **Diagram View**. Reordering automatically updates the message link numbers.
- Change the **Sequence Number** field in the Object Inspector.
- In the **Diagram View**, use the in-place editor to change the sequence number.

To specify creation of an object with a message:

1. Select a message link in the Sequence diagram.
2. In the Object Inspector of the message link, click the **Creation** field.
3. Choose *True* from the list box.

Result: The message link points to the recipient object icon rather than to its lifeline. The created object moves downward along the lifeline to show that it exists at a point later in time from its creator.

By default, the Creation property is set to False in the Properties Window.

To specify destruction of an object with a message:

1. Select a message link in the Sequence diagram.
2. In the Object Inspector of the message link, click the Destruction field.
3. Choose True from the list box.

Result: The object is destroyed.

By default, the **Destruction** property is set to *False* in the Object Inspector.

To specifying a return link by using the Tool Palette (Toolbox):

1. Click the **Return link** button in the Tool Palette.

2. On the sequence diagram, click the object lifeline element at the supplier end of the message link to draw the return link.

To specify a return link by using the Object Inspector (Properties Window):

1. Select the message link on the sequence diagram.
2. Choose View | Object Inspector on the main menu or press `F4`.
3. In the Object Inspector, click the drop-down arrow for the **Return Arrow** field and select *True*.

See Also

Rerouting a Link (see page 204)

UML 1.5 Interaction Diagram (see page 1131)

2.9.4.40 Designing a UML 1.5 Statechart Diagram

Following are tips and techniques that you can use when working with UML 1.5 Statechart Diagram.

To design a UML 1.5 Statechart Diagram, follow this general procedure:

1. ?reate Start and End points.
2. Create main states and substates.

   Tip: You can create nested states.

3. Create transitions.
4. Create history nodes.
5. You can optionally create shortcuts to related elements of other diagrams.

To create entry and exit actions:

1. Create an internal transition in the desired state.
2. Double-click the internal transition to enable in-place editing.
3. Rename using the following syntax: `stereotype/actionName(argument)` For example: `exit/setState(idle)`

Alternatively, create an internal transition and set the event name, event arguments, and action expression properties using the Object Inspector for the internal transition.

See Also

Creating an Internal Transition (see page 235)

Creating a Shortcut (see page 208)

UML 1.5 Statechart Diagram Reference (see page 1135)

2.9.4.41 Creating a Pin

To add an input pin, output pin, or value pin, do one of the following:

1. Right-click an action.
2. Choose **New** ▶ **Input Pin (or: Output Pin, or: Value Pin)** on the context menu.

Result: The created pin is added to the target action as a square. Note that the pins are attached to their actions, and can be only dragged along the action borders.

Alternatively:

1. Open the Tool Palette.
2. Choose the appropriate button, and click the target action.

See Also

Pin (see page 1141)

2.9.4.42 Designing a UML 2.0 Activity Diagram

Use the following tips and techniques when you design a UML 2.0 Activity Diagram. Usually you create Activity Diagrams after State Machine Diagrams.

To design a UML 2.0 Activity Diagram, follow this general procedure:

1. Create one or more activities. You can place several activities on a single diagram, or create a separate diagram for each.

   Warning: You cannot create nested activities.

2. Usually activities are linked to states or transitions on State Machine Diagrams. Switch to your State Machine Diagrams and associate the activities you just created with states and transitions.

   Tip: After that you can find that some more activities must be created, or the same activity can be used in several places.

3. Switch back to the Activity Diagram. Think about flows in your activities. You can have an object flow (for transferring data), a control flow, both or even several flows in each activity.
4. Create starting and finishing points for every flow. Each flow can have the following starting points:

- Initial node
- Activity parameter (for object flow)
- Accept event action
- Accept time event action Each flow finishes with a **Activity Final** or **Flow Final** node. If your activity has several starting points, they can be used simultaneously.

5. Create object nodes. You do not link object nodes to classes on your Class Diagrams. However, you can use hyperlinks for better understanding of your diagrams.
6. Create action nodes for your flows. Flows can share actions.

   Warning: You cannot create nested actions.

7. For object flows, add pins to actions. Connect actions and pins by flow links.
8. Add pre- and postconditions. You can create plain text or OCL conditions.
9. You can optionally create shortcuts to related elements of other diagrams.

To add an activity parameter to an activity:

1. In the Tool Palette, press the **Activity Parameter** button.
2. Click the target activity. Or: Choose **Add** ▶ **Activity Parameter** on the activity context menu.

Result: An **Activity Parameter** node is added to the activity as a rectangle. Note that the activity parameter node is attached to its activity. You can only move the node along the activity borders.

Note: Activity parameters cannot be connected by control flow links.

See Also

Associating a Transition or a State with an Activity (see page 230)

Grouping Actions into an Activity (see page 219)

Creating a Shortcut (see page 208)

UML 2.0 Activity Diagram Reference (see page 1140)

2.9.4.43 Grouping Actions into an Activity

Use the following techniques to group actions into an activity:

1. Use the Tool Palette buttons
2. Use drag and drop
3. Use the context menu of the activity element

Use the name="Delphi"Tool Palette buttons:

1. In the diagram Tool Palette, choose to create an activity node.
2. Choose the action button, and click the target activity.

Use drag and drop:

1. Place an action element on the diagram background.
2. Drag and drop the new action on top of an existing activity.

Use the context menu of the activity element:

1. Right-click the target activity.
2. Select **New ▶ Action** on the context menu.

See Also

Designing UML 2.0 Activity Diagram (see page 218)

UML 2.0 Activity Diagram Reference (see page 1140)

2.9.4.44 Working with an Object Flow or a Control Flow

You can create control flow or object flow as an ordinary link between the two node elements. The valid nodes are highlighted when the link is established.

You can scroll to the target element if it is out of direct reach, or you can use the context menu command to avoid scrolling.

There are certain limitations stipulated by UML 2.0 specifications:

- Object flow link must have an object at least on one of its ends.
- It is impossible to connect two actions with an object flow except through an output pin on the source action.
- Control flow link may not connect objects and/or activity parameters.

Use the following techniques with an object flow or a control flow:

1. Create a flow
2. ?reate a fork or a join

3. ?reate a decision or a merge

To create a flow:

1. Right-click the source element of the flow.
2. On the context menu, choose **Add ▶ Control Flow** or **Add ▶ Object Flow**. The **Choose Destination** dialog box opens.
3. In the **Choose Destination** dialog box, select the target and click OK. Note that the OK button is only enabled when the valid target is selected.

To create a fork or a join:

1. Identify the actions involved. If necessary, place all of the actions on the diagram first. Lay them out as desired.
2. Place either a fork or a join on the diagram. Resize as needed.
3. If depicting multiple sources, draw control flow from each of the source actions to the join, and from the join to the target action. If depicting multiple targets, draw control flow from the source action to the fork, and from the fork to each of the target actions.

To create a decision or a merge:

1. Identify the actions involved. If necessary, place all of the actions on the diagram first. Lay them out as desired.
2. Place either a decision or a merge on the diagram. Resize as needed.
3. If merging multiple actions, draw control flow from each of the source actions to the merge, and from the merge to the target action. If making a decision, draw control flow from the source action to the decision, and from the decision to each of the target actions.

See Also

Creating a simple link (see page 209)

UML 2.0 Activity diagram (see page 1140)

2.9.4.45 Designing a UML 2.0 Component Diagram

Following are tips and techniques that you can use when working with UML 2.0 Component Diagrams. It can be convenient to start creation of a model with Component Diagrams if you are modeling a large system. For example, a distributed, client-server software system, with numerous interconnected modules. You use Component Diagrams for modeling a logical structure of your system, while you use Deployment Diagrams for modeling a physical structure.

To design a UML 2.0 Component Diagram, follow this general procedure:

1. Create a hierarchy of components. The largest component can be the whole system or its major part (for example, *server application*, *IDE*, *service*).

   Tip: You can create nested component nodes. There are two methods for creating a nested component node: You can select an existing component and add a child component inside. Alternatively, you can create two separate components and connect them with an Association-Composition link.

2. In the hierarchy of components, you can end up by adding concrete classes and instance specifications. You can create them on a Component Diagram directly, or create them on a Class Diagram and put shortcuts on a Component Diagram.
3. Create interfaces. Each component can have a provided interface and a required interface.
4. Optionally, create artifacts. Usually, you describe physical artifacts of your system on Deployment Diagrams. But if some component is closely connected with its physical store, add and link an artifact to a Component Diagram.

   Tip: You can create nested artifacts.

5. Optionally, create ports for your components. You can attach a port to a component and link it with several classes or components inside. In this case, when a message arrives, this port decides which class must handle it.
6. Draw links between elements.
7. You can optionally create shortcuts to related elements of other diagrams.

See Also

Working with a Provided or Required Interface (see page 238)

Creating a Shortcut (see page 208)

UML 2.0 Component Diagram Reference (see page 1145)

2.9.4.46 Creating a Delegation Connector

To create a delegation connector:

1. Right-click an interface and choose **New ▶Delegation connector** from the context menu.
2. In the **Choose Destination** dialog box that opens, select the target interface from the Model or Favorites.
3. Click **OK**.

See Also

UML 2.0 composite structure diagram (see page 1146)

2.9.4.47 Creating an Internal Structure for a Node

To create an internal structure for a node:

1. Choose the part icon on the diagram Tool Palette.
2. Click the valid container (class or collaboration).
3. Repeat these steps to create as many participants as needed.

   Tip: Choose the part icon on the diagram Tool Palette

   while holding down the `CTRL` key. Each click on a valid container produces a new part.
4. Link the collaborating parts by connectors.
5. Use the Object Inspector to set up the properties of the part.

See Also

UML 2.0 composite structure diagram (see page 1146)

2.9.4.48 Creating a Referenced Part

To create a referenced part:

1. Open the **Diagram View**.
2. Do one of the following:

- Use the referenced part button on the diagram Tool Palette.
- Right-click a target container and choose **New ▶Referenced part** on the context menu.
- Select a part, open the **Model View**, and check the option aggregated by reference.

See Also

UML 2.0 composite structure diagram (see page 1146)

2.9.4.49 Creating a Port

To create a port:

1. Choose the port icon on the Tool Palette.
2. Click the target class or part.
3. Create as many ports as required.

See Also

UML 2.0 composite structure diagram (see page 1146)

2.9.4.50 Working with a Collaboration Use

To create a collaboration use:

1. On the diagram Tool Palette, choose the Collaboration Use button.
2. Click the target container.
3. Specify the name of the Collaboration Use.

To link to a collaboration type:

1. Select a Collaboration Use element.
2. Specify the type of Collaboration Use using one of the following methods:

- In the type field of the Collaboration Use in the Object Inspector, click the chooser button, and select the collaboration, which the Collaboration Use instantiates, from the Model or Favorites.
- Next to the name of the Collaboration Use, insert a colon and the name of the collaboration, which the Collaboration Use instantiates.

Result: The type of collaboration use is indicated next to its name.

To unlink from a collaboration type:

1. Right-click the Collaboration Use that has a certain type assigned.
2. On the context menu, choose Unlink Collaboration.

To bind with a role (part):

1. On the diagram Tool Palette, choose the Role Binding button.
2. If you hover the mouse over the client collaboration use, the valid client is highlighted with a black ellipse.
3. Drag-and-drop the role binding link to the supplier part. The valid target is highlighted.
4. Type the role name and press Enter to close the in-place editor.

If a collaboration use is associated with a collaboration that contains parts (roles), you can bind them with the parts (roles) of another classifier.

To bind the roles (parts) of the different classifiers via the collaboration use:

1. Create a collaboration use and define its type.
2. Create one or more parts in the collaboration that represents the type.

3. Right-click the target collaboration use and choose Bind new role on its context menu.
4. In the Select Destination dialog box that opens, choose the role to be bound in the target classifier.

Result: A role link is created from the collaboration use to the role in the target classifier. The role link is now marked with the name of the role selected in the collaboration.

Note: Each role can be used for binding only once. With the next invocation of the Bind new role command, the list of available roles no longer displays the ones previously used.

To define an owner:

1. Right-click a collaboration use and choose Object Inspector on its context menu.
2. In the owning classifier field of the Object Inspector, click the chooser button.
3. In the **Select Owning Classifier** dialog box, navigate to the owner class or collaboration and click OK.

Result: A link is created between the owner as supplier, and the collaboration use as the client. The link is marked with the label `<<represents>>`.

See Also

UML 2.0 composite structure diagram (↗ see page 1146)

2.9.4.51 Designing a UML 2.0 Deployment Diagram

Use the following tips and techniques when you design a UML 2.0 Deployment Diagram. It can be convenient to start creation of a model with Deployment Diagrams if you are modeling a large system that is comprised of multiple modules, especially if these modules reside on different computers. You use Deployment Diagrams for modeling a physical structure of your system, while you use Component Diagrams for modeling a logical structure.

To design a UML 2.0 Deployment Diagram, follow this general procedure:

1. Create a hierarchy of execution environments, devices, and nodes. Execution environments usually represent software environment used to execute your system, such as an operating system. Devices usually represent hardware equipment, such as a printer, a hard disk, or a computer. Nodes represent the rest of physical entities, such as a file.

   Tip: You can create nested execution environments, devices, and nodes. For example, you can add a node inside of an execution environment, or a node inside of a device.

2. Create artifacts.
3. Create deployment and instance specifications. By doing this, you arrange physical locations of objects and other entities of your system.
4. Add operations to artifacts.
5. Once an operation is added, you can define its properties in the Object Inspector, which includes parameters, stereotype, multiplicity and more.
6. You can optionally create shortcuts to related elements of other diagrams.

To deploy an artifact to a target node:

1. In the diagram Tool Palette, choose the deployment button.
2. Click the artifact to be deployed. The valid source is denoted by a solid frame.
3. Drag-and-drop the deployment link to a target node. The valid target is denoted by a solid frame.

To define parameters of an operation:

1. Select the desired operation in an artifact.

2. In the Object Inspector, expand the **General** node and choose **Parameters** field.
3. Click the chooser button to open **Add/Remove Parameters** dialog box.
4. Click **Add**. This creates an entry in the parameters list.
5. Enter the parameter's name, type multiplicity, default value, and direction. Note that parameter type can be selected from the list of pre-defined types, or from the model.
6. Using the **Add** and **Remove** buttons, create the list of parameters.
7. Click **OK** when ready.

See Also

Creating a Shortcut (↗ see page 208)

UML 2.0 Deployment Diagram Reference (↗ see page 1148)

2.9.4.52 Associating a Lifeline with a Classifier

To associate a lifeline with a classifier:

1. Select a lifeline on an Interaction diagram.
2. Right-click the lifeline and select **Choose▶Type...** on the context menu. The **Choose represented connectable element's type** dialog box opens.
3. Choose a classifier to be associated with the lifeline from the tree of available model elements.
4. Click **OK**.

See Also

Instantiating a classifier (↗ see page 211)

UML 2.0 Interaction diagram (↗ see page 1149)

2.9.4.53 Copying and Pasting an Execution or Invocation Specification

Clipboard operations are supported for the execution and invocation specifications.

To copy and paste an execution or invocation specification:

1. Cut, Copy, and Paste commands are available on the context menu of an execution specification and invocation specification. It is possible to copy or move these elements within the same diagram or to another diagram.
2. When an execution or invocation specification is copied, it means that the entire branch of messages is copied also. Pasting the clipboard contents to a target lifeline results in changing the message numbers according to the numbering of messages in the target lifeline.
3. If you paste an invocation or execution specification to another diagram, the entire outgoing bunch of messages will be pasted also, with all the respective lifelines. If the target diagram does not contain lifelines for this execution specification, they will be created automatically.

   Tip: It is also possible to move and copy message branches using the drag-and-drop technique. To move an execution or invocation specification, drag-and-drop it to the target location. To create a copy, drag-and-drop while holding the CTRL

   key down.

See Also

Working with UML 2.0 messages (↗ see page 227)

UML 2.0 interaction diagrams (see page 1149)

2.9.4.54 Creating a Sequence or Communication Diagram from an Interaction

To create a sequence or a communication diagram from an interaction:

1. In the **Model View**, choose an Interaction element.
2. Right-click the Interaction node and choose Open with Sequence diagram 2.0 or Open with Communication diagram 2.0.

Results: If such diagram is missing, it will be created. Then this diagram opens in the **Diagram View**.

See Also

UML 2.0 Interaction diagram (see page 1149)

2.9.4.55 Creating a State Invariant

To create a state invariant as an OCL comment:

1. On the UML 2.0 Sequence Diagram Tool Palette, choose the **State Invariant** button.
2. Click the target lifeline or execution specification.

   Tip: Alternatively, use the Add->State invariant

   command on the context menu of a lifeline or an execution specification.
3. In the Object Inspector of the state invariant, select the **General** node.
4. In the Invariant kind field, choose **OCL expression** from the drop-down list. The shape of the state invariant diagram element changes to braces.
5. In the OCL invariant node that adds to the Object Inspector, select the language of the comment from the **Language** drop-down list. The possible options are OCL and plain text.
6. Type the text and apply changes.

To connect a state invariant to a state:

1. On the diagram Tool Palette, choose the **State Invariant** button.
2. Click the target lifeline or execution specification.
3. In the Object Inspector of the state invariant, select the **General** node.
4. In the **Invariant kind** field, choose **States/Regions** from the drop-down list.
5. In the **States/Regions** field, click the chooser button.
6. In the **Choose States and/or Regions** dialog box, select the desired states and/or regions from the model, using the **Add** button.
7. Click **OK** when ready.

   Tip: Alternatively, type the state or region name. If the state or region belongs to a different package, specify its fully-qualified name.

See Also

OCL Support Overview (see page 95)

UML 2.0 Interaction Diagrams (see page 1149)

2.9.4.56 Designing a UML 2.0 Sequence or Communication Diagram

Use the following tips and techniques when you design a UML 2.0 Sequence or Communication Diagrams. Usually you create Interaction Diagrams after Class Diagrams.

Whenever an interaction diagram is created, the corresponding interaction is added to the project. Interactions are represented as nodes in the **Model View**.

Note: Presentation of an interaction in the Model View

depends on the view type defined in the **Model View** options on the default or project group levels. If model-centric mode is selected, an interaction is shown both under its package node and diagram node. If diagram-centric mode is selected, an interaction is shown under the diagram node only.

Note: You can view an interaction in two ways: as a Sequence Diagram, or as a Communication Diagram. So doing, any actions performed with either view are automatically reflected in the other views. Thus, adding or deleting an element in an interaction results in the modification of the corresponding interaction diagram, and vice versa. An interaction diagram contains a reference to the underlying interaction.

Note: Unlike UML 1.5, it is not possible to switch a diagram that already exists from sequence to communication and vice versa. However, it is possible to create a Sequence Diagram and a Communication Diagram based on the same interaction.

To design a UML 2.0 Sequence Diagram, follow this general procedure:

1. Create an interaction use
2. Navigate to a referenced interaction
3. Associate a lifeline with a referenced element
4. Associate a lifeline with a type
5. Define decomposition for a lifeline
6. Repeat the steps to create all required interactions
7. Link the created lifelines by using messages

To create an interaction use:

1. In the diagram Tool Palette, choose the **Interaction Use** button.
2. Click on the target lifeline.

   Tip: Alternatively, use the Add command on the lifeline context menu in the Diagram View

   or **Model View**.
3. In the Object Inspector for the newly created interaction use, choose the Properties tab.
4. In the interaction name field, click the chooser button.

   Tip: Alternatively, just type in the interaction name.

5. In the **Choose Referenced Interaction** dialog box, select the desired interaction from the project or Favorites, and click **OK**.

An interaction use is initially created attached to a lifeline. Further it can be expanded over several lifelines, detached from and reattached to lifelines.

To navigate to a referenced interaction:

1. Right-click on an interaction use that refers to another interaction.
2. On the context menu, choose Select.

3. Choose the desired destination on the submenu.

To associate a lifeline with a referenced element:

1. Make sure that your project contains the referenced elements that should be represented by the lifelines.
2. Select the desired lifeline in the **Model View** or the **Diagram View**.
3. In the Object Inspector, select the represents field.
4. Click the chooser button.
5. In the **Choose Represented Connectable Element** dialog box, select the desired part from the project or Favorites.
6. Click **OK**.

To associate a lifeline with a type:

1. Select the desired lifeline in the **Model View** or the **Diagram View**.
2. In the Object Inspector, select the type field.
3. Click the chooser button.
4. In the **Choose Represented Connectable Element's type** dialog box, select the class that defined the type from the project or Favorites.
5. Click **OK**.

To define decomposition for a lifeline:

1. Select the desired lifeline in the **Model View** or the **Diagram View**.
2. In the Object Inspector, select the decomposition field.
3. Click the chooser button.
4. In the **Choose Referenced Interaction** dialog box, select the desired interaction from the project or Favorites.
5. Click **OK**.

   Tip: Decomposition, type, stereotype, and referenced element properties are also reflected in the corresponding Communication diagram.

See Also

UML 2.0 Interaction Diagram Reference (see page 1149)

2.9.4.57 Linking Another Interaction from an Interaction Diagram

To link another interaction from an interaction diagram:

1. Open an Interaction diagram.
2. Right-click the diagram background and choose **Add ▸ Shortcut** on the context menu.
3. Add a shortcut to another interaction in your project.

See Also

UML 2.0 Interaction diagram (see page 1149)

2.9.4.58 Working with a UML 2.0 Message

This section describes techniques for working with messages in sequence and communication diagrams. Although the two diagram types are equivalent, the techniques for dealing with messages differ.

Use the following technique for UML 2.0 messages:

1. Show or hide reply message
2. Nest messages
3. Create a message from a lifeline back to itself
4. Create a message link that corresponds to an operation call

To show or hide reply message:

1. Select a call message in an interaction diagram.
2. In the Link tab of the Object Inspector, check or clear show reply message.

To nest messages:

1. You can nest messages by originating message links from an execution specification. The nested message inherits the numbering of the parent message. For example, if the parent message has the number 1, its first nested message is 1.1.
2. It is also possible to create message links back to the parent execution specifications.

To create a message from a lifeline back to itself:

1. Click the **Message** button on the Tool Palette.
2. In a Sequence diagram, click twice the lifeline in the place where you want this message to appear. In a Communication diagram, click twice the lifeline anywhere.

To create a message link that corresponds to an operation call:

1. Create an interaction.
2. Create a message link between two lifelines in the interaction.
3. Open the **Link** tab of the message link Object Inspector.
4. In the signature field, click the browse button.
5. In the Model or Favorites, select the desired operation.
6. Click **OK**.

The message link is named according to the name of the operation.

See Also

Working with an Instance Specification (↗ see page 237)

UML 2.0 Interaction Diagram (↗ see page 1149)

UML 2.0 Message (↗ see page 1153)

Execution and Invocation Specification (↗ see page 1151)

2.9.4.59 Working with a Combined Fragment

To create a combined fragment:

1. Choose the combined fragment button in the diagram Tool Palette, and click on the target lifeline.
2. In the **Type Chooser** dialog box that opens, choose the desired operator from the list of available operators.

Alternatively, you can also create a combined fragment using the context menu of the **Model View**, or **Diagram View**.

To do this, choose the desired lifeline or execution specification in the **Model View**, or in the **Diagram View**. On the context menu of the selection, choose **Add ▸ Combined Fragment**. This adds a combined fragment to the target location.

Result: the combined fragment is added to the target lifeline or execution specification. Each new combined fragment has different color, to tell it from the other combined fragments within the same cluster of nested frames.

To create a nested operator:

1. Select the desired combined fragment.
2. In the Operators field of the Object Inspector, click the chooser button. Edit **Combined Fragment Operators** dialog box opens.
3. In the Edit operator combobox, select the desired operator. If a certain operator enables parameters, enter the parameter values in the adjacent field. Use commas as delimiters.
4. Click **Add** button. A new line displays below the existing entry in the list of operators, and in the descriptor of the combined fragment.
5. Use **Add** and **Remove** buttons to make up the desired list of the nested operators. Use **Up** and **Down** buttons to specify the proper order of nested operators.
6. Click **Done** to apply changes.

Result: the nested operators are listed in the descriptor of the combined fragment in the specified order.

You can create the nested combined fragments by placing a new combined fragment node inside of an existing one. So doing, each new node is displayed in a different color. The colors are selected at random. You can work with the inner frames same way as with the outer frames: move along a lifeline, spread them over several lifelines, detach and tie frames. Note that drawing a message link from a frame automatically expands it, together with its outer frames, if any.

To create an operand:

1. Select the desired combined fragment in the **Model View** or in the **Diagram View**.
2. On the context menu of the combined fragment, choose **Add ▸ Interaction operand**.
3. In the Interaction constraint node select the language to be used for describing constraint. To do this, click the Language drop-down list and choose OCL or plain text.
4. Type the constraint expression.
5. Add as many operands as required.
6. Apply changes.

Result: a new operand is created. Constraint text is displayed in the operand section of the combined fragment.

See Also

OCL Support Overview (see page 95)

UML 2.0 Interaction Diagram (see page 1149)

2.9.4.60 Working with a Tie Frame

To spread a frame to several lifelines:

1. In the diagram Tool Palette, choose the Tie Frame button.
2. Click the desired interaction use or combined fragment.
3. Drag-and-drop on the target lifeline.

Result: The frame expands to the target lifeline and is attached to it with a dot.

See Also

UML 2.0 Interaction diagram (see page 1149)

2.9.4.61 Associating a Transition or a State with an Activity

You can associate an activity (created on some UML 2.0 Activity Diagram) with a state (on entering the state, while doing the state activity, and on exiting the state), or with a transition between states.

To associate a transition with an activity:

1. Select a transition or a state on a UML 2.0 State Machine diagram.
2. Under the General node of the Object Inspector, click the **Effect** (for a transition) or **Do activity**, **Entry** or **Exit** (for a state) field.
3. Click the chooser button to open the **Choose Activity** dialog box.
4. In the model treeview, locate the desired activity.
5. Click **OK**.

Tip: Once a guard condition or effect are specified in the Object Inspector

, you can further edit them in the diagram by double-clicking the expression to activate the in-place editor.

See Also

Creating a Guard Condition for a Transition (↗ see page 230)

UML 2.0 State Machine Diagram Reference (↗ see page 1155)

State (↗ see page 1117)

2.9.4.62 Creating a Guard Condition for a Transition

To create a guard condition for a transition:

1. Select a transition in the diagram.
2. Under the General node of the Object Inspector, click the **Guard** field.
3. Type the condition expression and apply changes.

See Also

OCL support overview (OCL expression) (↗ see page 95)

UML 2.0 State Machine diagrams (↗ see page 1155)

2.9.4.63 Creating a History Element

To create a history element for a state:

1. In the target state on a state diagram, select the target region where history needs to be added.
2. Choose **Shallow History** or **Deep History** on the diagram Tool Palette.
3. Click the target region.

See Also

History (State Machine diagrams) (↗ see page 1155)

UML 2.0 State Machine diagram (↗ see page 1155)

2.9.4.64 Creating a Member for a State

To create a member for a state:

1. Open the **Diagram View**.
2. Right-click an existing state and choose **Add ▶ (member)** on the context menu. The following members are available:

- Internal transition
- Entry point
- Exit point
- Region

See Also

UML 2.0 State Machine Diagram (see page 1155)

2.9.4.65 Creating a State

To create a state:

1. Using the Tool Palette buttons: On the diagram Tool Palette, choose to create a state node. Click an appropriate place on your diagram. Alternatively: Using the context menu of the diagram: Right-click the diagram background. Select **Add ▶ State** on the context menu.

   Note: You can place a state inside of the existing state. It is possible to hide individual states. For example, you might want to hide the content of composite states for better understanding of the whole diagram.

2. When a new state is placed on a diagram, you can use the Object Inspector to adjust its properties, including:

- Configure standard properties of the element.
- In the State Invariant field, select the language of the expression from the Language list box. The possible options are OCL and plain text.
- In the Properties page, configure the behavior of the state by setting or viewing the following additional properties:

Field	Description
Composite	Set to True if there is one or more regions in this state (not editable)
Orthogonal	Set to True if there are two or more regions in this state (not editable)
Simple	Set to True if there are no regions in this state (not editable)
Do activity	Specify the activity to be performed during execution of the current state by using the Object Inspector. This activity may be selected from any Activity diagram of the project
Entry	Specify the activity to be performed when the current state starts executing by using the Object Inspector. This activity may be selected from any Activity diagram of the project
Exit	Specify the activity to be performed when the current state finishes executing by using the Object Inspector. This activity may be selected from any Activity diagram of the project

In the edit field below the list box enter the OCL expression for this state.

See Also

OCL Support Overview (see page 95)

UML 2.0 State Machine Diagram (see page 1155)

2.9.4.66 Designing a UML 2.0 State Machine Diagram

Following are tips and techniques that you can use when working with UML 2.0 State Machine Diagram.

To design a UML 2.0 State Machine Diagram, follow this general procedure:

1. Create initial and final nodes.
2. Create main states and substates.
3. Create regions.
4. Create entry and exit points.
5. Create pins.
6. Create transitions.
7. Create history nodes.
8. You can optionally create shortcuts to related elements of other diagrams.

See Also

Creating a Shortcut (see page 208)

UML 2.0 State Machine Diagram Reference (see page 1155)

2.9.4.67 Browsing a Diagram with Overview Pane

To open the Overview pane:

1. Open a diagram and click the **Overview** button. The pane expands to show a thumbnail image of the current diagram.
2. Click the shaded area and drag it. This is a convenient way to scroll around the diagram.
3. Resize the **Overview** pane by clicking the upper-left corner of the pane and dragging it.
4. Close the **Overview** pane by clicking the diagram.

See Also

Zooming a diagram (see page 234)

2.9.4.68 Hiding and Showing Model Elements

You can control the visibility of elements on a diagram by using the Hide command (available on the context menu for individual diagram elements), and the Show/Hide command (available on the diagram context menu).

To hide by using one of the following methods:

1. Open the **Diagram View**.
2. Do one of the following:

- Select the element on the diagram, right-click and choose Hide on the context menu.
- Select multiple elements on the diagram using `CTRL+Click` or by lassoing, and select Hide from the context menu.
- Right-click the diagram background and choose Hide/Show on the context menu. The Show Hidden dialog box opens, as discussed below.

To show or hide diagram elements using the Show Hidden dialog box:

1. Right-click the diagram and choose Show/Hide on the context menu. The **Show Hidden** dialog box opens.
2. Select the element(s) that you wish to hide from the Diagram Elements list.
3. To add elements in the Diagram Elements list to the Hidden Elements list, do one of the following:

- Double-click the element .
- Click the element once and click Add.
- Select multiple elements using `CTRL+Click` and click Add.

4. To remove items from the Hidden Elements list do one of the following:

- Double-click the element.
- Click the element once and click **Remove**.
- Select multiple elements using `CTRL+Click` and click **Remove**.
- To remove all items from the Hidden Elements list, click **Remove All**.

5. Click **OK** to close the dialog box.

See Also

Using View Filters (↗ see page 233)

Creating a Single Element (↗ see page 209)

2.9.4.69 Using View Filters

For global control over the diagram view, you can use the filters in the **Options** dialog window.

To enable, disable view filters:

1. Choose **Tools**▶**Options** on the main menu.
2. Click the Together folder.
3. Under the **(level)**▶**Diagram** node, select View Filters.

   Note: The filters shown in the Options dialog window are global filters. To specifically filter classes, you can set the Show members property to False.

To filter classes:

1. In the **Options** dialog window, View Filters page, click the Show members field.
2. Click the drop-down arrow and select False.
3. Click **OK**.

This results in disabling the members, and the inner classifiers (classes, delegates, enumerations, interfaces, and structures).

Since inner classifiers are treated as members of the container element, the following filters do not filter inner classifiers:

View filter
Show classes
Show delegates
Show enumerations
Show interfaces

Show structures

Note: Code-specific elements are available in implementation projects only.

See Also

Hiding and Showing Model Elements (see page 232)

Together Diagram View Filters Options (see page 1096)

2.9.4.70 Zooming a Diagram

Use the diagram context menu to obtain the required magnification in the **Diagram View**.

To specify the magnification in the Diagram View:

1. Right-click the diagram background.
2. Select Zoom on the context menu.
3. Choose a command from the submenu.

See Also

Zoom keyboard shortcuts (see page 1104)

2.9.4.71 Working with a Complex State

The techniques in this section pertain to models of particularly complex composite states and substates.

You can resize the main state. You can also create a substate by drawing a state diagram within another state diagram and indicating start, end, and history states as well as transitions.

Create a composite state by nesting one or more levels of states within one state. You can also place start/end states and a history state inside of a state, and draw transitions among the contained substates.

Use the following techniques to create a composite (nested) state:

1. Create a nested substate using drag-and-drop
2. ?reate a nested substate using the context menu of the state element

To create a nested substate using drag-and-drop:

1. Place a state element on the diagram background.
2. Drag a new state on top of an existing state.
3. Drop a new state.

To create a nested substate using the context menu of the state element:

1. Right-click the state (region) that will be the container.
2. Select **Add ▶ State** on the context menu.

   Tip: You can nest multiple levels of substates inside one state. For especially complex substate modeling, however, you can find it more convenient to create different diagrams, model each of the substate levels individually, and then hyperlink the diagrams sequentially.

   Using the Shortcuts command on the context menu of the diagram, you can reuse existing elements in other state diagrams. Right-click the diagram and choose Add > Shortcuts, navigate within the pane containing the tree view of the available project contents for the project group

solution to the existing diagram, and select its elements, states, histories, forks, and/or joins.

Tip: Using the context menu of the state element, you can also create all of the other subelements that a state can contain.

Tip: Only one History element can be created within one state.

See Also

Hyperlinking Overview (see page 92)

Creating a Shortcut (see page 208)

UML 1.5 Activity Diagram (see page 1117)

UML 1.5 Statechart Diagram (see page 1135)

UML 2.0 State Machine Diagram (see page 1155)

2.9.4.72 Creating a Deferred Event

You can add a deferred event to a state element.

To create a deferred event:

1. Select the desired state or activity element on the diagram or in the **Model View**.
2. Right-click the element, and select **Add** ▶ **Deferred Event** on the context menu.

See Also

Deferred event (see page 1117)

UML 1.5 Activity diagram (see page 1117)

UML 1.5 Statechart diagram (see page 1135)

2.9.4.73 Creating an Internal Transition

To create an internal transition:

1. Select the desired state or activity element on the diagram or in the **Model View**.
2. Right-click the element, and select **Add** ▶ **Internal Transition** on the context menu.

See Also

Creating an Multiple Transition (see page 235)

Transition (see page 1118)

UML 1.5 Activity diagram (see page 1117)

UML 1.5 Statechart diagram (see page 1135)

UML 2.0 State Machine Diagram (see page 1155)

2.9.4.74 Creating a Multiple Transition

To create a multiple transition (a fork or a join):

1. Identify the states involved. If necessary, place all of the states on the diagram first and arrange as desired.

2. On the diagram Tool PaletteToolbox choose the fork or join button.
3. Place either a horizontal or vertical fork or join on the diagram.
4. Resize as needed.
5. On the diagram Tool PaletteToolbox, choose the transition button.
6. Draw links from the source state(s) to the fork/join node, and from the fork/join node to the target state(s).

See Also

Creating an Internal Transition (see page 235)

Transition (see page 1118)

UML 1.5 Activity diagram (see page 1117)

UML 1.5 Statechart diagram (see page 1135)

UML 2.0 State Machine Diagram (see page 1155)

2.9.4.75 Creating a Self-Transition

To create a self-transition:

1. Draw a transition from the state or activity element and drag the link away from the element.
2. Drag the link back to the element and drop it.

Alternatively:

1. Draw a transition between two activities (or states).
2. Drag the opposite end of the link line back to the desired activity (or state).

See Also

Creating a Simple Link (see page 209)

UML 1.5 Activity Diagram (see page 1117)

UML 1.5 Statechart Diagram (see page 1135)

Tool Palette (see page 1114)

2.9.4.76 Specifying Entry and Exit Actions

You can create entry and exit actions as nodes, or as stereotyped **internal transitions**.

To specify entry and exit actions using the in-place editor:

1. Create an internal transition in the desired state.
2. Double-click the internal transition to enable in-place editing.
3. Rename the internal transition using the following syntax:

```
stereotype/actionName(argument)
```

For example:

```
exit/setState(idle)
```

To specify entry and exit actions using internal transitions:

1. Create the internal transition.
2. Set the event name, event arguments, and action expression properties using the Object InspectorProperties Window for the internal transition.

See Also

UML 1.5 Activity Diagram (↗ see page 1117)

2.9.4.77 Working with an Instance Specification

You can instantiate a classifier using the Object InspectorProperties Window or the in-place editor.

Use the following techniques with an instance specification:

1. Instantiate a classifier using the Object InspectorProperties Window
2. Instantiate a classifier using the in-place editor
3. Define the features of an instance specification
4. Add a slot to an instance specification element
5. Associate a slot with a structural feature
6. Set the slot value
7. Define the slot stereotype

To instantiate a classifier using the name="Delphi"Object Inspector name="TVS"Properties Window:

1. Select an instance specification in your diagram.
2. In the General node of the Object InspectorProperties Window, select the instantiates field.
3. Click the chooser button.
4. In the **Choose Class or Interface for Type** dialog box, select the classifiers from the available contents, using the **Add** and **Remove** buttons.
5. Click **OK** when ready.

To instantiate a classifier using the in-place editor:

1. Select an instance specification in your diagram.
2. Press `F2` to open the in-place editor. Alternatively, click twice on the instance specification name.
3. Type the name of an existing classifier, delimited by a colon, next to the instance specification name. For example, `InstanceSpecifcation1:Class1`.
4. Press `Enter`.

To define the features of an instance specification:

1. Insert slots into an instance specification element.
2. Associate the slots with the attributes of the instantiated classifiers.
3. Set value, and define the slot stereotype.

To add a slot to an instance specification element:

1. Add an instance specification element to your diagram.
2. Right-click the instance specification element on your diagram and choose **New ▶ Slot** on the context menu.

To associate a slot with a structural feature:

1. Select a slot in an instance specification element.
2. Expand the **General** node of the Object InspectorProperties Window.
3. In the defining feature field, click the chooser button.
4. In the **Choose Attribute for Defining Feature** dialog box, select the desired attribute and click **OK**.

To set the slot value:

1. Choose a slot.
2. Do one of the following:

- In the Object InspectorProperties Window of the slot, type the desired string in the value field.
- Invoke the in-place editor for the slot and type the value next to the slot name, delimited by a equal sign.

To define the slot stereotype:

1. In the Object InspectorProperties Window of the slot, expand the **General** node.
2. In the **Stereotype** filed, enter the stereotype value.

See Also

UML 2.0 Class Diagram (see page 1143)

UML 2.0 Interaction Diagram (see page 1149)

UML 2.0 Component Diagram (see page 1145)

UML 2.0 Composite Structure Diagram (see page 1146)

2.9.4.78 Working with a Provided or Required Interface

To create a provided interface:

1. Create class and interface node elements using the and Tool PaletteToolbox buttons.
2. On the diagram Tool PaletteToolbox, click the provided interface button.
3. Click the client class and drag the mouse to the interface node.

To create a required interface:

1. Create class and interface node elements using the and Tool PaletteToolbox buttons.
2. On the diagram Tool PaletteToolbox, click the required interface button.
3. Click the client class and drag the mouse to the interface node.

See Also

Changing appearance of interfaces (see page 242)

UML 2.0 Class diagram (see page 1143)

UML 2.0 Component diagram (see page 1145)

UML 2.0 Composite Structure diagram (see page 1146)

2.9.4.79 Creating an Association Class

To create an association class:

1. On the diagram Tool PaletteToolbox, select the association class button.
2. Click the diagram background. This adds a regular class icon for the association class, connected with the diamond icon.
3. Create participant classes.
4. Using the association end button, connect the n-ary association with the participant classes.

Result: The source code of an association class contains appropriate tags for the association class itself, and for each of the association end classes.

To delete an association class:

1. Right-click an association end link, association class, or connector.
2. Choose Delete or Delete from View on the context menu.

Result: The whole association class construct is deleted from the diagram.

See Also

Class Diagram Relationships (see page 1123)

UML 2.0 Class Diagram (see page 1143)

UML 1.5 Class Diagram (see page 1121)

2.9.4.80 Creating an Inner Classifier

This section includes instructions for adding inner classifiers to classes (including Windows classes, such as Windows forms, Inherited forms, User Controls and so on), structures, and modules (collectively, containers) in implementation projects.

You can add inner classifiers to class diagram elements (containers) using the respective context menu for the diagram element in the Diagram or **Model View**s. You can also select a classifier in the Tool PaletteToolbox and click the container element in the **Diagram View** to add the inner classifier to the container element.

Note: Modules are specific to Visual Basic projects.

Structure elements are available for implementation projects only.

Tip: You can use drag-and-drop or clipboard operations to remove an inner classifier from the container element.

To create an inner classifier by using the context menu:

1. Right-click the container element.
2. Choose **Add ▶ (Inner_classifier_type)**, where (Inner_classifer_type) is defined in the table above.

Using cut, copy, and paste:

1. Use the clipboard operations to either cut or copy an existing classifier.
2. Select the container element.
3. Use the clipboard operations to paste the selected classifier into the container element.

Using drag-and-drop:

1. Select an existing classifier in the **Diagram View**.

2. Drag-and-drop it onto a pre-existing container in the **Diagram View**. A blue border highlights the location that Together recognizes as a valid destination for dropping the inner classifier.

See Also

Creating a Single Element (see page 209)

UML 1.5 Class Diagram (see page 1121)

UML 2.0 Class Diagram (see page 1143)

Inner Classifiers (see page 1124)

2.9.4.81 Using a Class Diagram as a View

Class diagrams can also be used to create subviews of the project.

To use a class diagrams as a view:

1. Create a new class diagram.
2. Create shortcuts to the original diagram to easily and quickly build subset views for easier management.

   Tip: Using this feature, you can create views of distributed classes into one diagram, with Together automatically displaying any relationships that the gathered classes may have with each other.

   Note: In implementation projects, changes made here also update the source code, keeping diagram and source code in sync.

See Also

LiveSource Overview (see page 93)

UML 1.5 Class diagram (see page 1121)

UML 2.0 Class diagram (see page 1143)

2.9.4.82 Working with an Interface

This topic describes how to create and hide an interface on a class diagram.

To create an interface:

1. Create a class and an interface node elements using the and Tool PaletteToolbox buttons.
2. On the diagram Tool PaletteToolbox, click the **Generalization** link button.
3. Click the client class and drag the mouse cursor to the interface node.

To hide an interface:

1. Select an interface.
2. Right-click and choose **Hide** on the context menu.

   Tip: You can hide all interfaces by disabling the Show Interfaces

   view filter.

See Also

Changing Appearance of Interfaces (see page 242)

UML 1.5 Class Diagram (see page 1121)

UML 2.0 Class Diagram (see page 1143)

2.9.4.83 Working with a Relationship

You can change the type of an association link.

To draw an association link:

1. Use the association link button on the UML Class Diagram Tool PaletteToolbox to draw association links between diagram elements.
2. The Object InspectorProperties Window enables you to set the link type (association, aggregation, or composition) and the cardinality of the client and supplier.
3. You can also set the link type using the right-click menu of the link. When you create an association link, Together defines a field in the client class (the start of the link).

To set the directed property of an association link:

1. Choose View | Object InspectorProperties Window if the Object InspectorProperties Window is not open.
2. Select a link on the diagram. The properties for the link appear in the Object InspectorProperties Window.
3. In the Object InspectorProperties Window, select the Directed field.
4. Click the drop-down arrow and select the Directed property (*True* or *False*) from the list.

See Also

Creating a simple link (see page 209)

Changing type of a link (see page 199)

UML 1.5 class diagrams (see page 1121)

UML 2.0 class diagrams (see page 1143)

2.9.4.84 Adding a Member to a Container

You can add members to class diagram elements (containers) by using the respective context menu for the diagram element in the **Diagram** or **Model Views** or available shortcut keys to add members to a class diagram container element.

To add a member to a container:

1. Right-click the container (class, interface, and so on).
2. Choose **Add ▶ (Member_type)**, where, Member_type, is defined in the table above.

   Tip: You can also use keyboard shortcuts to add fields and methods (functions in Visual Basic projects)

   to a container allowing such members. Click `CTRL+W` (for fields) and `CTRL+M` (for methods, functions).
3. You can edit the member using the in-place editor, Object InspectorProperties Window, or source code editor.

Result: The new member is placed in the compartment of the container in the sort order for the elements in your diagrams. You can set the sort order in the **Options** dialog window.

Tip: If a container already has members, you can right-click the existing member to create an additional member using the context menu. You can also select the member, and press INSERT

.

See Also

Creating a single element (see page 209)

Members (available to add) (↗ see page 1125)

UML 1.5 Class diagrams (↗ see page 1121)

UML 2.0 Class diagrams (↗ see page 1143)

2.9.4.85 Changing Appearance of Compartments

You can collapse or expand compartments for the different members of class, interface, namespace, module (Visual Basic projects only), enum, and structure (C# projects only) elements. By default, the compartments for these elements are displayed on the diagram as a straight line. You can use the **Options** dialog window to set viewing preferences for compartment controls. Adding compartment controls is particularly useful when you have large container elements with content that does not need to be visible at all times.

To collapse or expand compartments:

1. Select the class (or interface) on the diagram.
2. Click the “**+**” or “**-**” in the left corner of the compartment.

To view the compartment controls:

1. Open the Options dialog window.
2. Select the **Together** ▶ **(level)** ▶ **Diagram** ▶ **Appearance** ▶ **Nodes** category.
3. In this category, edit the **Show compartments as line field**.

See Also

UML 2.0 Class diagrams (↗ see page 1143)

UML 1.5 Class diagrams (↗ see page 1121)

Diagram Appearance options (↗ see page 1089)

2.9.4.86 Changing Appearance of Interfaces

To show an interface as a circle sing the context menu:

1. Right-click the interface element in the **Diagram** or **Model View**s.
2. Choose Show as circle.

   Tip: This menu item works as a toggle. Right-click again and choose Show as circle to show the interface element as a rectangle.

   Note: Interfaces shown as small circles do not show their members in the Diagram View

   . Use the **Model View** to view the members.

To show an interface as a circle using the name="Delphi" name="ide"Object Inspector name="TVS" name="ide"Properties Window:

1. Select the interface element in the **Diagram** or **Model View**s.
2. Press F4 to open the Object InspectorProperties Window.
3. Set the **Circle view** property as True.

   Tip: Choose False for the Circle view

   property to show the interface element as a rectangle.

See Also

Changing notation (↗ see page 197)

2.9.4.87 Working with a Constructor

You can create as many constructors in a class as needed.

In design projects, a constructor is created as an operation with the `<<constructor>>` stereotype.

In implementation projects, each new constructor is created with its unique set of parameters. In addition to creating parameters automatically, you can define the custom set of parameters, using the Object InspectorProperties Window.

Tip: You can move, copy and paste constructors and destructors between the container classes same way as the other members.

To define the constructor parameters (implementation projects only):

1. Select the desired constructor in a class.
2. In the Object InspectorProperties Window, click the Params field.
3. In the text field, type the list of parameters in the former type name. Use comma as a delimiter.

See Also

UML 2.0 Class diagrams (↗ see page 1143)

UML 1.5 Class diagrams (↗ see page 1121)

2.9.4.88 Working with a Field

This topic applies to implementation projects only.

In the source code, it is possible to declare several fields in one line. This notation is represented in diagram as a number of separate entries in the Fields section if a class icon. However, you can rename the fields, change modifiers, set initial values and so on, all modifications being applied to the respective field in the diagram icon. Also you can copy and move such fields in diagram (using context menu commands or drag-and-drop), and the pasted field appears in the target container separately.

To rename a field:

1. Choose a field.
2. Enter the new name in the in-place editor of the **Diagram View** or **Model View**, Name text field in the Object InspectorProperties Window or the source code editor.

To define the visibility modifier:

1. Choose a field.
2. Enter the visibility symbol in the in-place editor in the **Diagram View**, or select one from the Visibility combobox in the Object InspectorProperties Window, or edit in the source code editor.

To define the stereotype of a field:

1. Choose a field.
2. Use the in-place editor in the **Diagram View**, or stereotype combobox of the Object InspectorProperties Window or the source code editor.

To define modifiers, initial values, associated objects and so on:

1. Choose a field.
2. Use the Object InspectorProperties Window or the source code editor.

So doing, the model and the source code are kept in sync.

See Also

Synchronizing the Model View (see page 267)

Creating a Single Element (see page 209)

UML 1.5 class diagrams (see page 1121)

UML 2.0 class diagrams (see page 1143)

2.9.4.89 Associating a Message Link with a Method

Message links can be associated with the methods of the recipient class. The methods can be selected from the list of existing ones or can be created. This is done by two commands provided by the message context menu: Add and Choose method.

You can use the **Operation** field in the Object InspectorProperties Window to rename the method. A dialog box appears asking if you want to create a new method or rename the old one.

Use the following techniques to associate a message link with a method (operation):

1. Create a new method for an existing message link
2. Associate an existing method with a message link
3. Unlink a method

To create a new method for an existing message link:

1. Create a message link between two objects. The recipient object must instantiate a class.
2. On the context menu of the message link, choose Add. The submenu provides the choice of Method, Constructor or Destructor.

   Note: Destructors are available for classes in C# projects only.

3. From the submenu, choose the required operation type.

   Tip: If the recipient object does not instantiate a class, the Add command is not available on the context menu.

   If the recipient object is associated with an interface, only methods can be associated with the message link.

   Result: The new operation is created in the class of the recipient object. The message link is labeled with the operation name, according to the operation type:

   If a Method is selected, the label is `Method<n> ():return_type`.

   If a Constructor is selected, the label is `<Classname>()` in C# projects and `<New>()` in Visual Basic .NET projects.

   If a Destructor is selected, the label is `~<Classname>()`. The Destructor option is disabled in the submenu of the Add command.

   You can use the **Operation** field in the Object InspectorProperties Window to create a new method in the classifier. For example, in the Operation field, you can enter `method_name(parameter_types):return_type`. Entering `parameter_types` is optional. Entering the `return_type` is optional for Visual Basic .NET projects. If the method does not exist in the class, a dialog opens prompting you to create it. If the method already exists in the class, the message link is automatically set for that method.

To associate an existing method with a message link:

1. Create a message link between two objects. The recipient object must instantiate a class.
2. On the context menu of the message link, select Choose method. The submenu displays the list of operations of the recipient class.
3. If you cannot find the required operation in the list, click More to reveal the next 20 methods (including inherited operations) of the recipient class.
4. Select the required operation.

Result: The associated operation is selected from the list of available methods, constructor, or destructor.

If you choose to associate a different classifier for an object that is already instantiated with a classifier, all of the message links where the **Operation** property has been set are automatically saved as text unless the method signature matches another method signature within the newly-linked classifier.

To unlink a method:

1. Select the message link.
2. On the context menu of the message link, choose Unlink method.

Result: An association between the message link and the operation is removed. However, the operation is preserved in the recipient class.

If you unlink a classifier from an object and that object has incoming message links where the **Operation** property is set to a method of the unlinked classifier, a dialog opens prompting you to unlink the method from the message link or save it as text. Choosing the option to save as text places the **Operation** property in quotation marks and the operation displays in red on the diagram. The intent of this feature is to help users to preserve all of the signatures of any methods that have been linked to the message links. Upon instantiating the object with a class again, you can delete the quotation marks. This will open a dialog box prompting you to create the method if it does not exist in the linked classifier. A dialog box does not open if the signature of the method matches an existing method in the classifier.

See Also

Creating a simple link (↗ see page 209)

UML 1.5 interaction diagrams (↗ see page 1131)

2.9.4.90 Generating an Incremental Sequence Diagram

You can generate incremental sequence diagrams from a previously-generated sequence diagram. In some cases, you can have generated a sequence diagram with a low nesting value such as 3 or 5. The nesting value limits how deep the parser traverses the source code calling sequence.

To generating an incremental sequence diagram from a previously-generated sequence diagram:

1. Once you review the sequence diagram, you can decide that you want to see additional objects and messages that are currently not shown on the diagram because of the nesting value constraint.
2. In this case, you can select the Generate Sequence Diagram command from the context menu of an activation block and the nested messages and objects calling from that method display on the diagram.

See Also

Roundtrip Engineering for Interaction Diagrams

Creating a Shortcut (↗ see page 208)

UML 1.5 Interaction Diagrams (↗ see page 1131)

2.9.4.91 Creating a Browse-Through Sequence of Diagrams

You can link entire diagrams at one level of detail to the next diagram up or down in a sequence of increasing granularity, or you can link from key use cases or actors to the next diagram.

To create a browse-through sequence:

1. Think of the answers to the following questions:

- Why do you want to link several diagrams? What is your general idea?
- What types of diagrams are you going to link? Usually you link Use Case Diagrams, but you can include any other types to a sequence.
- Do you want to create "vertical" top-down links, or "horizontal" links for diagrams at one level?
- Do you want to link entire diagrams, or specific model elements?

2. Open the main diagram of the sequence you are going to create.
3. Select the source model element, or right-click the diagram background to link the entire diagram.

   Note: If you choose to hyperlink to a new diagram, its shortcut appears on the source diagram.

   Tip: It is recommended to use some common approach for all links in your sequence.

4. Create a hyperlink to the next diagram. The titles of source and destination elements turn blue.
5. Open the destination diagram.
6. Repeat steps 3–5 for all parts of your sequence.
7. Optionally, create hyperlinks in the reverse motion.

See Also

Hyperlinking Overview (↗ see page 92)

UML 1.5 Use Case Diagram (↗ see page 1137)

UML 2.0 Use Case Diagram (↗ see page 1157)

2

2.9.4.92 Creating an Extension Point

To create an extension point:

1. Right-click the use case element.
2. Choose **Add ▶ Extension Point** on the context menu.
3. Type in a name.

See Also

UML 1.5 use case diagram (↗ see page 1137)

UML 2.0 use case diagram (↗ see page 1157)

2.9.4.93 Designing Use Case Hierarchy

Use case diagrams typically represent the context of a system and system requirements.

To design use case hierarchy:

1. Usually, you begin at a high level and specify the main use cases of the system.
2. Next, you determine the main system use cases at a more granular level. As an example, a "Conduct Business" use case can have another level of detail that includes use cases such as "Enter Customers" and "Enter Sales."
3. Once you have achieved the desired level of granularity, it is useful to have a convenient method of expanding or contracting the use cases to grasp the scope and relationships of the system's use case views.

See Also

UML 1.5 Use Case Diagram (see page 1137)

UML 2.0 Use Case Diagram (see page 1157)

2.9.5 Together Documentation Generation Procedures

This section provides how-to information on using Together Documentation Generation facilities.

Topics

Name	Description
Configuring the Documentation Generation Facility (see page 247)	To define the documentation title, header, footer and other specific settings, use the **Options** dialog window. Descriptions of the options are provided in the **Options** dialog window. You can also find their descriptions in this online help.
Generating Project Documentation (see page 248)	

2

2.9.5.1 Configuring the Documentation Generation Facility

To define the documentation title, header, footer and other specific settings, use the **Options** dialog window.

Descriptions of the options are provided in the **Options** dialog window. You can also find their descriptions in this online help.

To configure the documentation generation facility:

1. On the main menu, choose **Tools** ▶ **Options** ▶ **Together** ▶ **(level)** ▶ **Generate Documentation**.
2. Under the General category, enter the documentation title, window title, header, and footer.
3. Set the User Internal Browser option to choose to open the generated documentation in an external browser or in the RAD Studio internal browser. By default, documentation opens in your external browser.
4. Under the Include category, select the visibility modifiers for classes and members to be included in the generated documentation.
5. Under the Navigation category, set up the options for generating navigation bar, index, class hierarchy, and help link.

See Also

Documentation Generation Facility Overview (see page 100)

Generating Project Documentation (see page 248)

Together Generate Documentation Options Reference (see page 1099)

2.9.5.2 Generating Project Documentation

To generate project documentation:

1. Select project name, namespace or diagram in the **Model View**.
2. Select Tools->Generate Documentation on the main menu. Alternatively, right-click the selection and choose Generate Documentation on the context menu.
3. In the **Generate Documentation** dialog box that opens, select your preferred Scope and Options settings.
4. Click **OK** to generate documentation. By default, the Generate Documentation wizard creates documentation for your entire project.

See Also

Documentation Generation Facility Overview (see page 100)

Configuring the Documentation Generation Facility (see page 247)

Together Generate Documentation Options Reference (see page 1099)

2.9.6 Using Online Help

To get assistance while you work, do one of the following:

1. To see a description of what any screen element does in any opened dialog box, press `F1` or click **Help**.
2. To see a relevant help topic for a pane, view, Tool Palette icon or another element, press `F1`.
3. To open the Table of Contents for online help, choose Help ->CodeGear Help on the main menu to see the **Contents** tab.
4. To search for specific topics and terms, use the **Index** tab.
5. If you have questions about RAD Studio, visit CodeGear Technical Support at http://support.borland.com.

   Tip: To filter out the unnecessary books and topics from the Table of Contents and index, choose one of the filters in the Filtered By

   list box. If a topic provides information that can be relevant to one or another RAD Studio feature set, you can show or hide the desired contents within a topic using the filter button.

See Also

Help on Help (see page 51)

Keyboard Mappings (see page 1068)

Together Keyboard Shortcuts (see page 1104)

2.9.7 Together Object Constraint Language (OCL) Procedures

This section provides how-to information on using Together OCL facilities.

Topics

Name	Description
Creating an OCL Constraint (see page 249)	
Editing an OCL Expression (see page 249)	
Showing and Hiding an OCL Constraint (see page 250)	

2.9.7.1 Creating an OCL Constraint

To create an object constraint and link it with the context:

1. In the Class/package diagram Tool Palette, choose the Constraint button and click the diagram background. The note element appears with the OCL editor activated.
2. Type the constraint expression.
3. Close the OCL editor.
4. In the diagram Tool Palette, choose the button, and link the constraint node with the respective design element.

   Tip: The constrained attribute should actually exist in the context. Otherwise the constraint will be marked as invalid.

Alternatively, follow these steps:

1. In the **Model View** or in the diagram, right-click an element for which a constraint should be created.
2. Choose Constraints.
3. In the Add / Remove constraints dialog box, click Add.
4. Enter the constraint:

- In the Name field, enter the constraint name.
- In the Language field, choose OCL or text from the list box.
- In the Constraint field, enter the constraint text.

5. Add as many constrains as needed.
6. Click **OK** when ready.

See Also

OCL Support Overview (see page 95)

Working with Combined Fragments (see page 228)

OCL Editor (Diagram View) (see page 1106)

2.9.7.2 Editing an OCL Expression

To activate the OCL Editor:

1. Double-click a constraint element or OCL property, or select a constraint element and press F2. The OCL Editor window opens.
2. Edit an expression.
3. Use the green button to apply changes and close the OCL Editor. Use the red button to discard changes and close the OCL Editor.

See Also

OCL Support Overview (see page 95)

Working with combined fragments (see page 228)

Creating an OCL constraint (see page 249)

OCL editor (Diagram View) (see page 1106)

2.9.7.3 Showing and Hiding an OCL Constraint

To hide an individual constraint:

1. Right-click a constraint in the diagram.
2. Choose Hide.

To hide multiple constraints:

1. Right-click the diagram background.
2. Choose Show/Hide.
3. In the Show Hidden dialog box, select the desired constraints in the Diagram Elements list.
4. Click Add.

To reveal the hidden constraints:

1. Right-click the diagram background.
2. Choose Show/Hide.
3. In the Show Hidden dialog box, select the desired constraints in the Hidden list.
4. Click Remove.

See Also

OCL Support Overview (see page 95)

Working with combined fragments (see page 228)

Creating an OCL constraint (see page 249)

Editing OCL expression (see page 249)

OCL editor (Diagram View) (see page 1106)

2.9.8 Working with a Namespace or a Package

Namespaces are used in implementation projects, and packages in design projects.

Use the following techniques for a namespace or a package:

1. View a namespace or a package
2. Open a namespace or a package
3. Delete a namespace or a package
4. Rename a namespace or a package

To view a namespace or a package:

1. By default, a namespace element on a diagram displays the namespace contents.
2. You can use the context menu of a class or interface in a namespace to add fields and methods directly.

To open a namespace or a package:

1. Choose the Open Diagram command on the namespace diagram context menu.
2. You can also double-click the namespace element on the diagram.

To delete a namespace or a package:

1. Open the **Diagram View** or the **Model View**.
2. Choose Delete on its context menu.

   Warning: Deleting a namespace also deletes all of its contents.

To rename a namespace or a package:

1. Open a project.
2. To rename a namespace, including changing the namespace name in all of its source files, do one of the following:

- Choose Rename on the context menu of a namespace in the **Diagram View** or in the **Model View**
- Invoke the in-place editor for the namespace element in the **Diagram View** or in the **Model View**
- Edit the Name field in the Object Inspector

See Also

Namespace and Package Overview (see page 89)

Creating a Project (see page 264)

2.9.9 Together Pattern Procedures

This section provides how-to information on using patterns with Together.

Topics

Name	Description
Adding Participants to the Patterns as First Class Citizens (see page 253)	Patterns as First Class Citizens are represented by the **GoF** patterns. When such patterns are applied, the elements are created with the standard number of participants. However, you can add allowed participants to the existing pattern object. If you add participants, links between the pattern object and the new participants are created.
Creating a Pattern (see page 253)	You can use existing diagram elements as the basis to create custom patterns. The newly created patterns are stored in the **Pattern Registry**. They become visible in the pattern tree of the **Pattern Organizer** and can be used to generate design elements in diagrams.
Deleting Patterns as First Class Citizens from the Model (see page 254)	You can delete elements of the patterns as First Class Citizens (GoF patterns), using both the **Diagram View** and the **Model View**. If you delete elements, they are removed from the diagram and from the model.

Using the Pattern Registry (see page 254)	The **Pattern Registry** is only available from the Pattern Organizer context menu, when you create a new shortcut, or assign a pattern to a shortcut. In the Pattern Registry you can filter patterns by category, metaclass, diagram type, language or status of registration. To open the **Pattern Registry**, do one of the following: • Right-click a folder and choose **New shortcut**. • Right-click a pattern shortcut and choose **Assign Pattern**.
Creating a Link by Pattern (see page 255)	Together makes it easy for you to apply patterns when creating links. To create links during modeling, you can use the **Link by Pattern** button in the diagram Tool Palette. The **Link by Pattern** button launches the **Pattern Wizard** dialog displaying the available patterns.
Creating a Model Element by Pattern (see page 255)	You can apply patterns explicitly using the **Node by Pattern** button in the Tool Palette or by using the right-click menu command Create by Pattern. Whenever you create an element on a diagram using one of the toolbar buttons, you are applying a default pattern that is connected to the selected button.
Using the Stub Implementation Pattern (see page 255)	
Exporting a Pattern (see page 257)	You can create patterns and export them to the specified location.
Importing a Legacy Pattern (see page 257)	You can reuse patterns created in the different versions of Together. Upon starting Together, the available storage is scanned for patterns, and all the encountered patterns are included in the Pattern Registry. However, they are not available for usage unless you manually create shortcuts to these patterns in the **Pattern Organizer**.
Sharing Patterns (see page 258)	You can store patterns in the shared locations, to facilitate team development. The **Pattern Organizer** enables access to the shared patterns if the paths to these patterns are included in the list of Shared Pattern Roots. being included in the list, patterns from the shared location become visible in the Custom Patterns node of the patterns tree.
Assigning Patterns to Shortcuts (see page 258)	You can associate a pattern with one or more shortcuts, located in the various virtual folders.
Copying and Pasting Shortcuts, Folders or Pattern Trees (see page 258)	
Creating a Folder in the Pattern Organizer (see page 259)	Use virtual folders to logically organize patterns in the pattern trees.
Creating a Shortcut to a Pattern (see page 259)	In the Pattern Organizer you are working with shortcuts, not with the actual patterns. Because of this, shortcuts to the same pattern may be included in several folders.
Creating a Virtual Pattern Tree (see page 259)	The **Pattern Organizer** enables you to logically organize patterns using virtual trees, folders and shortcuts. Under a tree node you can create virtual folders and shortcuts to patterns.
Deleting shortcuts, folders or pattern trees (see page 260)	
Editing Properties (see page 260)	Properties of the virtual trees, folders and shortcuts are displayed in the properties section of the **Pattern Organizer**. Using the toolbar buttons, you can choose the properties presentation: in expandable nodes, or in alphabetical order. The **Name** and **Visible** properties are editable. Changes are applied when the edited field looses the focus, or the Enter key is pressed. The node name in the tree view changes accordingly.
Opening the Pattern Organizer (see page 260)	The **Pattern Organizer** enables you to logically organize patterns (using virtual trees, folders and shortcuts), and view and edit the pattern properties.
Saving Changes in the Pattern Registry (see page 261)	If you have changed the contents of the **Pattern Registry** using the **Pattern Organizer** (created new shortcuts, exported or created shared folders), these changes are synchronized with the Registry automatically. When you close the **Pattern Organizer**, you are prompted to save changes. Each time you start Together, the contents of the available storage is scanned for patterns. The contents of the registry is synchronized with the actual availability of the pattern folders. If you have made changes to the patterns outside the Organizer, these changes will be synchronized when Together is started.
Sorting Patterns (see page 261)	While working with the **Pattern Organizer**, the logical trees, folders, and shortcuts may be displayed in an arbitrary order. You can sort nodes alphabetically within the container node, using the **Sort Folder** command.

Using the Pattern Organizer (see page 261)	The **Pattern Organizer** enables you to: • Create logical pattern trees and folders • Create shortcuts to patterns • Assign patterns to shortcuts • Copy, paste and delete trees, folders and shortcuts • Save changes in the Pattern Registry

2.9.9.1 Adding Participants to the Patterns as First Class Citizens

Patterns as First Class Citizens are represented by the **GoF** patterns. When such patterns are applied, the elements are created with the standard number of participants. However, you can add allowed participants to the existing pattern object. If you add participants, links between the pattern object and the new participants are created.

To add a participant to a GoF pattern:

1. Select the oval pattern element in the **Diagram View** or **Model View**
2. Right-click on the pattern element choose **Add** from the context menu. The submenu presents the list of allowed participants.
3. Choose the required participant from the submenu.
4. In the **Pattern Action Wizard**, specify the name of the new participant, and click **OK**.

   Tip: If the participant with the specified name already exists, it is reused.

See Also

Patterns overview (see page 96)

Pattern Organizer (see page 1107)

Pattern Registry (see page 1109)

2.9.9.2 Creating a Pattern

You can use existing diagram elements as the basis to create custom patterns. The newly created patterns are stored in the **Pattern Registry**. They become visible in the pattern tree of the **Pattern Organizer** and can be used to generate design elements in diagrams.

To create a pattern:

1. Select one or more elements on a diagram.
2. Right-click and choose **Save as Pattern** on the context menu of the selection. The **Create Pattern Wizard** opens.
3. On the **General** page of the wizard enter the following information:

- In the **File** field specify the target XML file name.
- In the **Name** field specify the name of the new pattern.
- Optionally, enter the pattern description in the **Description** field
- Optionally, check **Create Pattern Object** check box. Selecting this option allows you to use your pattern as a First Class Citizen. This means that an oval pattern element will display on your diagrams when applying the pattern.
- Click **Next**.

4. On the **Pattern Parameters** page of the wizard:

- Use the in-line editor to modify the parameters as required.
- Set the **Use Existent** property for the pattern. If this value is checked, existing elements on the diagram are reused when you apply the pattern. This means that whenever you apply a pattern, a new element is not created if there is an element with the same name and metatype in the target container . If you clear the**Use Existent** property, then new elements are created.
- Click **Next**.

5. In the **Select tree folder** page that displays the current patterns structure, choose the target folder, and click **OK**.

Result: The new pattern is added to the specified folder. This pattern is visible in the pattern tree and can be used to generate design elements.

See Also

Patterns overview (see page 96)

Create Pattern Wizard (see page 1159)

Pattern Organizer (see page 1107)

Pattern Registry (see page 1109)

2.9.9.3 Deleting Patterns as First Class Citizens from the Model

You can delete elements of the patterns as First Class Citizens (GoF patterns), using both the **Diagram View** and the **Model View**. If you delete elements, they are removed from the diagram and from the model.

To delete a GoF pattern with participants:

1. In the **Diagram View** or **Model View**, select the oval pattern element to be deleted.
2. On the context menu of the selection, choose the **Delete with Participants** command.
3. Confirm deletion.

See Also

Patterns overview (see page 96)

Pattern Organizer (see page 1107)

Pattern Registry (see page 1109)

2.9.9.4 Using the Pattern Registry

The **Pattern Registry** is only available from the Pattern Organizer context menu, when you create a new shortcut, or assign a pattern to a shortcut. In the Pattern Registry you can filter patterns by category, metaclass, diagram type, language or status of registration.

To open the **Pattern Registry**, do one of the following:

- Right-click a folder and choose **New shortcut**.
- Right-click a pattern shortcut and choose **Assign Pattern**.

To filter patterns in the Pattern Registry:

1. In the Filters section of the Pattern Registry dialog window, click the attribute to filter the patterns.
2. Select the desired value from the drop-down list.

See Also

Patterns overview (see page 96)

Patterns Organizer (see page 1107)

Patterns Registry (see page 1109)

2.9.9.5 Creating a Link by Pattern

Together makes it easy for you to apply patterns when creating links. To create links during modeling, you can use the **Link by Pattern** button in the diagram Tool Palette. The **Link by Pattern** button launches the **Pattern Wizard** dialog displaying the available patterns.

To create a link by pattern:

1. Click the **Link by Pattern** button in the diagram Tool Palette. The button stays down.
2. Click the source element on the diagram.
3. Drag to the destination element and drop when the second element is highlighted. The **Pattern Wizard** opens.
4. In the **Pattern Wizard** window, select the pattern that you want to apply for the new link, define its properties and click **Finish**.

See Also

Patterns overview (see page 96)

Creating a model element by pattern (see page 255)

2.9.9.6 Creating a Model Element by Pattern

You can apply patterns explicitly using the **Node by Pattern** button in the Tool Palette or by using the right-click menu command Create by Pattern. Whenever you create an element on a diagram using one of the toolbar buttons, you are applying a default pattern that is connected to the selected button.

To create model elements by pattern:

1. On the diagram Tool Palette, choose the **Node by Pattern** button.
2. Click the container, where you want to add an element by pattern. This can be either the diagram background or a node element. **Pattern Wizard** opens.

   Tip: Alternatively, right-click the target container and choose Create by Pattern

   on the context menu.
3. In the **Pattern Wizard** select the desired pattern, modify its properties and click **OK**.

See Also

Patterns overview (see page 96)

Creating a link by pattern (see page 255)

2.9.9.7 Using the Stub Implementation Pattern

To create an inheritance link with stub implementation using the Link by Pattern button:

1. Click the Link by Pattern button in the Tool Palette.

2. Click the source class and drag-and-drop the link to the destination class or interface. The Pattern Wizard dialog opens.
3. In the Pattern Wizard, expand the Standard folder and select Implementation link and stub.
4. Click OK to complete the stub implementation. The inheritance link is created and the stubs for the inherited methods are generated in the source class.

To create an inheritance link with stub implementation using the Node by Pattern button:

1. Click the Node by Pattern button in the Tool Palette.
2. Select the source class on the diagram. The Pattern Wizard opens.
3. In the Pattern Wizard, expand the Standard folder, and select Implementation link and stub.
4. In the Pattern Properties pane on the right of the Pattern Wizard, click the information button to the right of the Supplier field. The Select Supplier dialog opens.
5. Select the destination class or interface from the treeview of available contents and click Ok.
6. Click OK to complete the stub implementation and close the Pattern Wizard. The inheritance link is created and the stubs for the inherited methods are generated in the source class.

To create an inheritance link with stub implementation using the Create by Pattern context menu:

1. Right-click the source class on the diagram and choose Create by Pattern from the context menu. The Pattern Wizard opens.
2. In the Pattern Wizard, expand the Standard folder and select Implementation link and stub.
3. In the Pattern Properties pane on the right of the Pattern Wizard, click the information button to the right of the Supplier field. The Select Supplier dialog opens.
4. Select the destination class or interface from the treeview of available contents and click Ok.
5. Click OK to complete the stub implementation and close the Pattern Wizard. The inheritance link is created and the stubs for the inherited methods are generated in the source class.

   Note: You can find the Stub implementation pattern on the context menu of classes that inherit from an interface or an abstract class. This pattern is also available in the Pattern Wizard by clicking the Node by Pattern button in the Tool Palette

   , or by using the Create by Pattern context menu for a class. Use the Stub implementation pattern if you already have an inheritance/generalization link drawn on the diagram and you want to copy the methods to the source class.

To create a stub implementation using the class context menu:

1. Right-click a class that inherits from an interface or an abstract class.
2. Choose Stub Implementation from the context menu.

To create a stub implementation using the Node by Pattern button:

1. Click the Node by Pattern button in the Tool Palette.
2. Select the source class on the diagram. The Pattern Wizard opens.
3. In the Pattern Wizard, expand the Standard folder, and select Stub implementation.
4. Click OK to complete the stub implementation and close the Pattern Wizard. The stubs for the inherited methods are generated in the source class.

To create a stub implementation using the Create by Pattern context menu:

1. Right-click the source class on the diagram and choose Create by Pattern from the context menu. The Pattern Wizard opens.
2. In the Pattern Wizard, expand the Standard folder, and select Stub implementation.
3. Click OK to complete the stub implementation and close the Pattern Wizard. The stubs for the inherited methods are generated in the source class.

See Also

Patterns overview (↗ see page 96)

Pattern Organizer (see page 1107)

Pattern Registry (see page 1109)

2.9.9.8 Exporting a Pattern

You can create patterns and export them to the specified location.

To export a pattern:

1. In the **Pattern Organizer** window, expand the pattern tree and locate the folder to be exported.
2. Right-click the selected folder and choose **Export folder**.
3. In the **Select path to export** dialog box, navigate to the desired location, and click **Save**.

See Also

Creating a pattern (see page 253)

Pattern Organizer (see page 1107)Pattern Registry (see page 1109)

2.9.9.9 Importing a Legacy Pattern

You can reuse patterns created in the different versions of Together. Upon starting Together, the available storage is scanned for patterns, and all the encountered patterns are included in the Pattern Registry. However, they are not available for usage unless you manually create shortcuts to these patterns in the **Pattern Organizer**.

To reuse a custom pattern, follow this general procedure:

1. Copy your legacy patterns to the folder that stores patterns in your product installation folder.
2. After the product startup, Pattern Registry automatically registers all available patterns.
3. Open the **Pattern Organizer**,
4. In the **Pattern Organizer** window:

- Locate the target folder for the patterns in question, or create a new folder.
- Create a new shortcut.
- Assign the desired pattern to this shortcut.

5. Save changes.

See Also

Patterns overview (see page 96)

Create a new folder (see page 259)

Create a new shortcut (see page 259)

Assign the desired pattern to this shortcut (see page 258)

Saving changes in Pattern Organizer (see page 261)

Pattern Organizer (see page 1107)Pattern Registry (see page 1109)

2.9.9.10 Sharing Patterns

You can store patterns in the shared locations, to facilitate team development. The **Pattern Organizer** enables access to the shared patterns if the paths to these patterns are included in the list of Shared Pattern Roots. being included in the list, patterns from the shared location become visible in the Custom Patterns node of the patterns tree.

To create shared patterns:

1. Export the desired patterns to a shared location.
2. In the **Pattern Organizer**, click **Edit Shared Patterns Roots**. **Shared Patterns Roots** dialog opens.
3. In the List of Shared Patterns Roots, click **Add**. **Select Shared Pattern Tree** dialog opens.
4. In the **Select Shared Pattern Tree** dialog locate the folder that contains the desired patterns, select the `Shortcut Registry.xml` file and click **Open**. The path is added to the list of shared pattern roots.
5. Edit the list using **Add** and **Remove** buttons.
6. Click **OK** when ready.

See Also

Exporting patterns (see page 257)

Pattern Organizer (see page 1107)

Pattern Registry (see page 1109)

2.9.9.11 Assigning Patterns to Shortcuts

You can associate a pattern with one or more shortcuts, located in the various virtual folders.

To assign a pattern to a shortcut:

1. In the **Virtual pattern trees** section of the **Pattern Organizer**, select the desired shortcut.
2. Right-click and choose **Assign Pattern**. The **Pattern Registry** opens.
3. In the **Pattern Registry**, select the pattern to be assigned to the selected shortcut, and click **OK**.
4. In the **Properties** section of the**Pattern Organizer**, edit the shortcut name and visibility as required
5. Save the changes.

See Also

Creating a Shortcut to a Pattern (see page 259)Editing Pattern Properties (see page 260)

Pattern Organizer (see page 1107)

Pattern Registry (see page 1109)

2.9.9.12 Copying and Pasting Shortcuts, Folders or Pattern Trees

To copy and paste a folder or a shortcut:

1. In the **Virtual pattern trees** section, select a shortcut, folder or pattern tree to be copied.
2. Right-click the node and choose **Copy** on the context menu.

   Tip: Alternatively, press CTRL+C

3. Right-click the destination node and choose **Paste** on the context menu. Alternatively, press `CTRL+V`
4. Save changes.

See Also

Pattern Organizer (see page 1107)

Pattern Registry (see page 1109)

2.9.9.13 Creating a Folder in the Pattern Organizer

Use virtual folders to logically organize patterns in the pattern trees.

To create a new virtual folder in the Pattern Organizer:

1. In the **Pattern Organizer**, select the target node in the **Virtual pattern trees** section.
2. Right-click this node and choose **New Folder**. The *New Folder* node is added.
3. In the **Properties** section, edit the **Name** and **Visible** fields as required.

See Also

Creating a Shortcut to a Pattern (see page 259)Creating a Tree (see page 259)

Pattern Organizer (see page 1107)

Pattern Registry (see page 1109)

2.9.9.14 Creating a Shortcut to a Pattern

In the Pattern Organizer you are working with shortcuts, not with the actual patterns. Because of this, shortcuts to the same pattern may be included in several folders.

To create a new shortcut to a pattern:

1. In the **Pattern Organizer**, select the topmost target node.
2. Right-click this node and choose **New Shortcut**. The **Pattern Registry** opens.
3. In the **Pattern Registry**, select the pattern to be assigned to the new shortcut, and click **OK**
4. When the **Pattern Organizer** prompts you to save changes in the **Pattern Registry**, click **Yes**.

See Also

Creating a folder (see page 259)Creating a tree (see page 259)

Pattern Organizer (see page 1107)

2.9.9.15 Creating a Virtual Pattern Tree

The **Pattern Organizer** enables you to logically organize patterns using virtual trees, folders and shortcuts. Under a tree node you can create virtual folders and shortcuts to patterns.

To create a new pattern tree:

1. In the **Pattern Organizer**, select the topmost **Patterns** node.

2. Right-click this node and choose **New Pattern Tree**. The *New Pattern Tree* node is added.
3. In the **Properties** section, edit the **Name** and **Visible** fields as required.

See Also

Creating a Folder (see page 259)Creating a Shortcut to a Pattern (see page 259)

Pattern Organizer (see page 1107)

2.9.9.16 Deleting shortcuts, folders or pattern trees

To delete a node from the Pattern Organizer:

1. In the **Virtual pattern trees** section, select a shortcut, folder or pattern tree to be deleted.
2. Right-click the node and choose **Delete** on the context menu. Alternatively, press `DELETE` key
3. Save changes.

See Also

Pattern Organizer (see page 1107)

Pattern Registry (see page 1109)

2.9.9.17 Editing Properties

Properties of the virtual trees, folders and shortcuts are displayed in the properties section of the **Pattern Organizer**. Using the toolbar buttons, you can choose the properties presentation: in expandable nodes, or in alphabetical order. The **Name** and **Visible** properties are editable. Changes are applied when the edited field looses the focus, or the Enter key is pressed. The node name in the tree view changes accordingly.

To edit properties of a tree, shortcut or folder:

1. Select a node in the **Virtual pattern trees** section.
2. In the **Properties** section, edit the **Name** property, using the text field.
3. In the **Properties** section, edit the **Visible** property, using the drop-down list.

   Tip: The Visible

   property applies to shortcuts only. If **Visible** is set to Visible, the shortcut is displayed in the Pattern Wizard. Otherwise, it is not visible. If a folder does not contain any visible shortcuts, it is also hidden in the Pattern Wizard.
4. Save changes.

See Also

Pattern Organizer (see page 1107)

Pattern Registry (see page 1109)

2.9.9.18 Opening the Pattern Organizer

The **Pattern Organizer** enables you to logically organize patterns (using virtual trees, folders and shortcuts), and view and edit the pattern properties.

To open the Pattern Organizer:

1. On the main menu, choose Tools->Pattern Organizer.
2. Result: The **Pattern Organizer** window opens.

See Also

Pattern Organizer (↗ see page 1107)

Pattern Registry (↗ see page 1109)

2.9.9.19 Saving Changes in the Pattern Registry

If you have changed the contents of the **Pattern Registry** using the **Pattern Organizer** (created new shortcuts, exported or created shared folders), these changes are synchronized with the Registry automatically. When you close the **Pattern Organizer**, you are prompted to save changes. Each time you start Together, the contents of the available storage is scanned for patterns. The contents of the registry is synchronized with the actual availability of the pattern folders. If you have made changes to the patterns outside the Organizer, these changes will be synchronized when Together is started.

To save changes in the Pattern Registry:

1. In the **Pattern Organizer** click **Close** button. The dialog window opens prompting you to save changes in the pattern registry.
2. Click **Yes** to confirm.

   Tip: Alternatively, open the Pattern Registry

   dialog, and click **Synchronize**.

See Also

Pattern Organizer (↗ see page 1107)

Pattern Registry (↗ see page 1109)

2.9.9.20 Sorting Patterns

While working with the **Pattern Organizer**, the logical trees, folders, and shortcuts may be displayed in an arbitrary order. You can sort nodes alphabetically within the container node, using the **Sort Folder** command.

To sort patterns the Pattern Organizer:

1. In the **Virtual pattern trees** section, select the node to be sorted.
2. Right-click the node and choose **Sort Folder** on the context menu.
3. Save changes.

See Also

Pattern Organizer (↗ see page 1107)

Pattern Registry (↗ see page 1109)

2.9.9.21 Using the Pattern Organizer

The **Pattern Organizer** enables you to:

- Create logical pattern trees and folders
- Create shortcuts to patterns
- Assign patterns to shortcuts
- Copy, paste and delete trees, folders and shortcuts
- Save changes in the Pattern Registry

See Also

Patterns Overview (see page 96)

Saving Changes in the Pattern Organizer (see page 261)

Creating a Folder (see page 259)

Opening the Pattern Organizer (see page 260)

Creating a Virtual Pattern Tree (see page 259)

Creating a Shortcut to a Pattern (see page 259)

Assigning Patterns to Shortcuts (see page 258)

Copying and Pasting in Pattern Organizer (see page 258)

Deleting a Node from the Pattern Organizer (see page 260)

Editing Pattern Properties (see page 260)

Sorting the Pattern Organizer (see page 261)

Pattern Registry (see page 1109)

Pattern Organizer (see page 1107)

2.9.10 Together Project Procedures

This section provides how-to information on using Together projects.

Topics

Name	Description
Activating Together Support for Projects (see page 263)	This topic describes how to activate Together support for a project individually. **Tip:** You can also force Together to activate support automatically for all new or currently open projects by adjusting General options.
Creating a Project (see page 264)	
Exporting a Project to XMI Format (see page 264)	
Importing a Project in IBM Rational Rose (MDL) Format (see page 265)	**IMPORTANT:** For the MDL Import function to work, the Java Runtime Environment and the Java Development Kit must be installed, and the paths in the `jdk.config` file must correctly point to your JDK/JRE directory. See the first step below
Importing a Project Created in TVS, TEC, TJB, or TPT (see page 265)	Together supports full backward compatibility with the previous version. You can open your old projects in the regular way. You can also import projects created in other editions of Together. **Warning:** Diagrams in projects must be created in the common diagram format `.txv*`. The legacy diagram format `.df*` is not supported. **Warning:** Diagram elements must be embedded (created as filemates). Standalone design elements (SDE) are not supported.
Importing a Project in XMI Format (see page 266)	

Opening an Existing Project for Modeling (see page 267)	You can add modeling capabilities to an existing implementation project that was created without Together. When you open a project subdirectory from the **Model View** or **Diagram View**, Together reverse engineers the contents into a namespace diagram that shows the namespaces, classes, and interfaces and their interrelationships.
Synchronizing the Model View, Diagram View, and Source Code (see page 267)	Together provides constant synchronization between different aspects of your project: • Model hierarchy, presented in the **Model View** • Model graphical representation in the **Diagram View** • Source code (for implementation projects) **Tip:** You can also use the Reload function of the Model View to update an entire model, and the Refresh function of the **Diagram View**.
Transforming a Design Project to Source Code (see page 269)	This feature is available for UML 1.5 and UML 2.0 design projects.
Troubleshooting a Model (see page 269)	You can also reload your project from the source code.
Working with a Referenced Project (see page 270)	Your project can have a binary library whose content you may want to display in your diagrams. For example, you can show entities that reside in the `MSCorLib.dll` or other project references. Such resources exist for the project, but Together does not include them in the generated HTML documentation for the project. The **Model View** enables you to view class diagrams for references included in your projects. You can add references to your project using the Project Manager.

2.9.10.1 Activating Together Support for Projects

This topic describes how to activate Together support for a project individually.

Tip: You can also force Together to activate support automatically for all new or currently open projects by adjusting General options.

To activate Together support follow these steps:

1. Switch to a desired project or project groupsolution.
2. Choose **Project ▶ Together Support** on the main menu. Result: The **Model Support** dialog box opens showing the list of projects within the current project groupsolution.
3. In the **Model Support** dialog box, check the flags for those projects where you need modeling.
4. Click **OK**.

Result: The **Model View** displays the models for each of the selected projects. In the Project ManagerSolution Explorer, `ModelSupport_%PROJECTNAME%ModelSupport` folder is added to each of the selected projects.

To deactivate Together support, follow the above procedure, but uncheck the flags for those projects of a project groupsolution that do not need modeling.

See Also

Creating a Project (see page 264)

Opening an Existing Project for Modeling (see page 267)

Together General Options (see page 1098)

2.9.10.2 Creating a Project

To create a Together project:

1. On the main menu, choose File->New->Other. The **New Project** dialog box opens.
2. From the **Project Types** or the **Item categories** pane, choose the desired project category.
3. From the templates pane, choose the desired project template.
4. Enter the project name, location and other parameters as required by the **New Project** dialog box.
5. Click **OK**.
6. Follow the procedure provided by the **New Project Wizard**.
7. In the **Project from MDL** wizard, click the Add Folder button and choose the desired source folder from the file system. Use the Remove and Remove all buttons to make up the list of model files.

Result: A project of the selected type is created in the specified location.

For design project, `.bdsproj` file is created in the specified project root. The default package and diagram are created.

For implementation project, if Together support is enabled, `.bdsproj` file is created in the specified project root, the default namespace and diagram are created.

See Also

Opening a Project for Modeling (see page 267)

Creating a Namespace or a Package (see page 250)

Creating a Diagram (see page 196)

Supported Project Formats (see page 1116)

2.9.10.3 Exporting a Project to XMI Format

To export a project to XMI format:

1. In the **Model View**, right-click the root project node, and choose Export Project to XMI, or choose **File▶Export Project to XMI** on the main menu. The **XMI Export** dialog box opens.
2. In the Select XMI Type groupbox, select the xml/uml version you wish the file to support. You can select from the available XMI Type choices:

- XMI for UML 1.3 (Unisys Extension)
- XMI for UML 1.3 (Unisys Extension, Recommended for TCC), default value
- XMI for UML 1.3 (Unisys Extension, Recommended for IBM Rational Rose)

3. Click the drop-down arrow to select an appropriate XMI encoding requirement. The default value is UTF-8.
4. Specify the export destination. You can include the path as well as the name of the file (.xml) which will be created, or you can accept the default: `(project folder)\out\xmi\(project name).xml`
5. Click Export. If the destination directory does not exist, a confirmation dialog asks if you want to create it.
6. Click Yes.

Result: The created XML file is added to the specified location.

See Also

Import and export features overview (see page 100)

XMI Export dialog box (see page 978)

2.9.10.4 Importing a Project in IBM Rational Rose (MDL) Format

IMPORTANT: For the MDL Import function to work, the Java Runtime Environment and the Java Development Kit must be installed, and the paths in the `jdk.config` file must correctly point to your JDK/JRE directory. See the first step below

To create a design project on the base of an IBM Rational Rose (MDL) project:

1. Open `[RAD Studio]5.0/bin/plugins/mdlimport/jdk.config` and modify the `javahome` and `addpath` statements to reflect the relative path from the `jdk.config` file to your JKD/JRE installation. For example,

```
javahome ../../../../../../../../jdk1.4.2/jre
addpath ../../../../../../../../jdk1.4.2/lib/tools.jar
```

Note: The path to the JDK/JRE should be without quotation marks.

Save the file and open RAD Studio.

2. On the main menu, choose File->New->Other. The **New Project** dialog box opens.
3. From the **Project Types** pane, choose Design Project.
4. From the **Templates** pane, choose **Convert from MDL** template.
5. Enter the project name, location and other parameters as required by the **New Project** dialog box.
6. Click **OK**.
7. In the **Project from MDL** wizard, specify the source `.mdl`, `.ptl`, `.cat`, or `.sub` file using the **Add** button.
8. Specify the scale factor and conversion options.
9. Click **Finish**.

Result: A new design project is created in the specified location.

See Also

Import and Export Features Overview (see page 100)

Convert From MDL Wizard (see page 1160)

2.9.10.5 Importing a Project Created in TVS, TEC, TJB, or TPT

Together supports full backward compatibility with the previous version. You can open your old projects in the regular way.

You can also import projects created in other editions of Together.

Warning: Diagrams in projects must be created in the common diagram format `.txv*`. The legacy diagram format `.df*` is not supported.

Warning: Diagram elements must be embedded (created as filemates). Standalone design elements (SDE) are not supported.

The general procedure for importing a project created in name="Delphi"TVS, TEC, TJB, or TPT consists of the following steps:

1. Creating a new project in RAD Studio
2. Importing the model information into this project

To create a new project for import:

1. Choose File->New->Other on the main menu. The **New Project** dialog window opens.
2. Select the project template. Note that the project type should correspond to the type of the source project:

- For a UML 1.x design project, choose Design project->UML 1.5 Design ProjectTogether design project->UML 1.5 Together Design Project.
- For a UML 2.x design project, choose Design project->UML 2.0 Design ProjectTogether design project->UML 2.0 Together Design Project.

3. Enter the project name.

   Warning: The project name should be exactly equal to the source project name. Adjust the remainder of the settings on your own.

4. Click **OK** to create a project.
5. Close the project when it is created.

To import the model information:

1. Open Windows Explorer or any other file manager.
2. Copy all model files including subfolders from the source project to the `ModelSupport_%PROJECTNAME%` folder under your new project root. These files are located under `diagrams`, `ModelSupport` or `Model Folder` directories, depending on the version of Together.

   Note: For some projects

   these files are located in the same folders as the source code files. In this case you will have to pick out the modeling files manually. Basically, you need all files with `.txv*` and `.txa*` extensions.

3. If you have an implementation project and you need to keep your source code, copy it from the source project to the new one keeping the folder structure.
4. Open the project in RAD Studio. Open the Project Manager.
5. Choose the project root node.
6. Right-click and choose Add... on the context menu. The Add to Project dialog box opens. In this dialog box, choose the first source file from the `src` folder and click OK.
7. Repeat the last steps for all source and modeling files.

Result: RAD Studio processes your files. When completed, the imported project is displayed in the **Model** and **Diagram Views**.

See Also

Interoperability Overview

Sharing Model Information Between TCC/TAR and RAD Studio

2.9.10.6 Importing a Project in XMI Format

To import a project in XMI format:

1. Open a diagram or have the project root node selected in the **Model View**.

   Warning: The project must comply with the UML 1.5 specification.

2. In the **Model View**, right-click the root project node and choose Import Project from XMI, or choose **File ▶ Import Project from XMI** on the main menu. The **XMI Import** dialog box opens.
3. Browse for the source file.
4. Click **Import**.

   Tip: The recommended way to import a project from Together ControlCenter (TCC) or Together Architect (TAR) to RAD Studio is to use the common diagram format.

You can import a model created with IBM Rational Rose directly.

See Also

Import and Export Features Overview (↗ see page 100)

Importing a Project in IBM Rational Rose (MDL) Format (↗ see page 265)

2.9.10.7 Opening an Existing Project for Modeling

You can add modeling capabilities to an existing implementation project that was created without Together.

When you open a project subdirectory from the **Model View** or **Diagram View**, Together reverse engineers the contents into a namespace diagram that shows the namespaces, classes, and interfaces and their interrelationships.

To open an existing implementation project for modeling:

1. Make sure that Together support is activated.
2. On the main menu, choose File->Open Project.
3. In the **Open Project** dialog box, specify the project location.
4. Select the project or project group file.
5. Click **OK**.

Result: With Together support activated, opening existing implementation project automatically reverse engineers the existing source code into class diagrams.

See Also

Activating Together support (↗ see page 263)

Creating a project (↗ see page 264)

2.9.10.8 Synchronizing the Model View, Diagram View, and Source Code

Together provides constant synchronization between different aspects of your project:

- Model hierarchy, presented in the **Model View**
- Model graphical representation in the **Diagram View**
- Source code (for implementation projects)

  Tip: You can also use the Reload function of the Model View

  to update an entire model, and the Refresh function of the **Diagram View**.

You can navigate between the Model View, Diagram View, and source code by using the following techniques:

1. Navigate to a diagram from the **Model View** to the **Diagram View**
2. Navigate to a model element from the **Model View** to the **Diagram View**
3. Navigate from the **Diagram View** to the **Model View**
4. Navigate from a lifeline to its classifier in the **Model View** or a Class diagram
5. Navigate from source code to the **Model View**
6. Navigate from the **Model View** or **Diagram View** to source code (for implementation projects)

7. Edit a synchronized element

To navigate to a diagram from the Model View to the Diagram View:

1. In the **Model View**, right-click the diagram node.
2. Choose Open Diagram.

Alternatively, double-click the diagram node in the **Model View**.

To navigate to a model element from the Model View to the Diagram View:

1. Select a model element in the **Model View**.
2. Right-click and choose Select on Diagram on the context menu.

   Note: If this model element appears on several diagrams, choose a diagram on the submenu.

To navigate from the Diagram View to the Model View:

1. Right-click the selected element or diagram background in the **Diagram View**.
2. Choose Synchronize with Model View on the context menu.

To navigate from a lifeline to its classifier in the Model View or a Class diagram:

1. Right-click the selected lifeline on a UML 2.0 Sequence diagram in the **Diagram View**.
2. Choose **Select ▶ Type in Model View** to navigate to the classifier in the **Model View**, Or: Choose **Select ▶ Type on Diagram** to navigate to the classifier on a Class diagram in the **Diagram View**.

To navigate from source code to the Model View:

1. Right-click the line that contains the desired element.
2. On the context menu of the selection, choose Synchronize Model View.

Result: The corresponding element is highlighted in the **Model View**.

To navigate from the Model View or Diagram View to source code (for implementation projects):

1. Right-click a model element or a node member.
2. Choose Go to definition on the context menu.

   Note: This command is available for source code-generating elements.

   Result: Source code of the element in question opens in the Editor tab. The corresponding definition is highlighted.

   Tip: To open source code of an entire class or interface, double-click the element icon.

To edit a synchronized element:

1. Select an element in the **Diagram View** or **Model View**.
2. Edit the desired fields in the Object Inspector.

   Note: Alternatively, invoke the in-line editor in the Diagram View

   or **Model View**.

   Warning: Avoid using the Structure View

   or the Project Manager for modification of the model elements.

See Also

LiveSource Overview (see page 93)

Troubleshooting the Model (see page 269)

2.9.10.9 Transforming a Design Project to Source Code

This feature is available for UML 1.5 and UML 2.0 design projects.

To generate source code from a design project:

1. In the **Model View**, select a design project.
2. Right-click and choose Transform to source on the context menu.
3. In the **Choose Destination Project** dialog box, select the desired implementation project.
4. Check the **Use name mapping files for code generation** checkbox if required.
5. Click **Transform**.

Result: implementation code of the class diagrams that existed in the design project are added to the target language-specific project. The diagrams are also added to the target project. The diagram roots are preserved.

To insert source code to an implementation project:

1. In the **Model View**, select an implementation project.
2. Right-click and choose Transform code from design project on the context menu.
3. In the **Choose Source Project** dialog box, select the desired design project.
4. Check the **Use name mapping files for code generation** checkbox if required.
5. Click **Transform**.

Result: implementation code of the class diagrams that existed in the design project are added to the target implementation project. The diagrams are also added to the target project. The diagram roots are preserved.

See Also

Transformation to source code overview (↗ see page 94)

"Choose Source (Destination) Project" dialog box (↗ see page 962)

2.9.10.10 Troubleshooting a Model

You can also reload your project from the source code.

Use the following techniques to troubleshoot your model:

1. Refresh a model
2. Reload a model
3. Fix a model

To refresh a model:

1. Open the **Diagram View**.
2. Press `F6`.

To reload a model:

1. Open the **Model View**.
2. Right-click the project root node and choose Reload on the context menu.

   Note: Use the Reload command as a workaround for issues that might appear while making changes in Together that cause some elements on the diagram to stop responding, or if you get errors from Together, such as, `<undefined value>`.

Tip: Usually, when these problems occur, the elements also disappear from the RAD Studio Structure View

and the corresponding source code is underlined in blue in the RAD Studio Editor. Together cannot always properly handle such elements that become broken. To restore broken elements to a normal state, it is necessary to edit the code in the text editor according to the recommendation shown in the RAD Studio Editor. In these cases, it is best to refresh the model using Reload to prevent possible further misbehavior.

To fix a model:

1. For interaction diagrams: regenerate them from the source code.
2. For all types of diagrams: check that none of the necessary elements are hidden.

See Also

Synchronizing the Model View (↗ see page 267)

Reload Command (Model View) (↗ see page 1112)

2.9.10.11 Working with a Referenced Project

Your project can have a binary library whose content you may want to display in your diagrams. For example, you can show entities that reside in the `MSCorLib.dll` or other project references. Such resources exist for the project, but Together does not include them in the generated HTML documentation for the project.

The **Model View** enables you to view class diagrams for references included in your projects. You can add references to your project using the Project Manager.

To add a project to references:

1. In the Project Manager, expand the desired project node.
2. On the context menu of the References node, choose Add Reference.

   Tip: Alternatively, choose Project->Add Reference

   on the main menu.
3. In the **Projects** tab, select the projects to be referenced and click **Select**.
4. Click **OK** when ready.

Result: The **Choose Type to Instantiate** dialog box shows all referenced projects, making it possible to choose the desired classifiers from the different projects.

To view a diagram of a referenced project:

1. Open or create a class diagram.
2. Right-click the diagram background and choose **Add ▶ Shortcuts**. The **Edit Shortcuts** dialog box opens and displays the content available for the diagram and all content residing outside of the current namespace.
3. Choose the resource that you want to add from the tree view of available contents on the left of the dialog and click **Add >>**.
4. Repeat until you have added all of the resources that you want to show on the diagram.
5. Click **OK** to close the dialog box.

   Tip: If the Edit Shortcuts

   dialog box does not show the resource that you are looking for, it is probably not added as a reference to your project. Choose **Project ▶ Add Reference** on the main menu to add a project reference.

To view the MsCorLib.dll (a standard DLL added automatically to your projects):

1. Expand the **References** node and the `MsCorLib.dll` node in the **Model View**.
2. Right-click the default diagram and choose Open Diagram. The default diagram opens in the **Diagram View**. You can expand the Microsoft and System folders to view other class diagrams as well.

See Also

Creating a Shortcut (see page 208)

Instantiating a Classifier (see page 211)

2.9.11 Together Quality Assurance Procedures

This section provides how-to information on using Together Quality Assurance facilities.

Topics

Name	Description
Exporting Audit Results (see page 271)	Export audit results to an XML or HTML file to share them with team members or review them later.
Printing Audit Results (see page 272)	You can print the entire table of audit violations, or select specific rows and columns. **Warning:** This feature is available for implementation projects only.
Running Audits (see page 273)	Audits automatically check for conformance to standard or user-defined style, maintenance, and robustness guidelines. Before running audits, make sure that the code being audited is compilable. If your source code contains errors, or some libraries and paths are not included, audits might produce inaccurate results. **Warning:** This feature is available for implementation projects only.
Viewing Audit Results (see page 274)	When viewing audit results, you can compare and organize items in the results report. The results report is tightly connected with the diagram elements and the source code. Using the report, you can navigate to the specific location of the violation. **Warning:** This feature is available for implementation projects only.
Working with a Set of Audits (see page 274)	
Creating a Metrics Chart (see page 275)	You can create a chart in the **Metric Results Pane**. Metrics charts are created in temporary files which are deleted when the charts are closed. However, you can save graphical information in text files, export it to the desired graphical format, and include graphics in project.
Running Metrics (see page 276)	Before running metrics, make sure that the code being analyzed can be compiled. If your source code contains errors or some libraries and paths are not included, metrics might produce inaccurate results. **Warning:** This feature is available for implementation projects only.
Viewing Metric Results (see page 276)	
Working with a Set of Metrics (see page 277)	

2.9.11.1 Exporting Audit Results

Export audit results to an XML or HTML file to share them with team members or review them later.

To save the audit results in a separate file:

1. Select the rows of the table that you want to save. Do not select anything if you want to print the entire list.
2. Click the **Save** button on the toolbar.
3. In the **Save Audit Results** dialog box that opens, choose the scope of the results to export using the **Select View** list box:

- **All Results**: If the results are grouped, choosing All Results prints a report for all groups in the current tabbed page. If the results are not grouped, then all results export for the current tabbed page.
- **Active Group**:If the results are grouped, you can select a group in the current tabbed page, and the generated report contains the results from the selected group.
- **Selected Rows**: You can select single or multiple rows in the audit results report view. Choosing Selected Rows generates a report for such selections.

4. Each tabbed page can contain a list of audits (when the audits are ungrouped) or a group tree with a list of the selected group (when the audits are grouped).

   Note: Unless the results have been grouped using the Group by command, the Active Group option is not enabled in the dialog box.

   Tip: You can use CTRL+CLICK

   to select multiple rows.

5. In the Select Format list box, select the format for the exported file:

- **XML**: Generates an XML-based report.
- **HTML**: Generates an HTML-based report. Selecting HTML format activates the following check boxes:
- **Add Description**: This saves the audit descriptions in a separate folder with hyperlinks to the descriptions from the results file.
- **Launch Browser**: This option opens the generated HTML file in the default viewer.

6. Click **Save** to save the results in the specified location.

See Also

Quality Assurance facilities overview (↗ see page 98)

Viewing audit results (↗ see page 274)

2.9.11.2 Printing Audit Results

You can print the entire table of audit violations, or select specific rows and columns.

Warning: This feature is available for implementation projects only.

To print the list of audit violations:

1. Select the rows of the table that you want to print. Do not select anything if you want to print the entire list.

   Tip: You can select multiple rows using CTRL+CLICK

   .

2. Click the Print button on the Toolbar. The Print Audit dialog box opens.
3. Choose the scope of the results to print using the Select View list box:

- **All Results**: If the results are grouped, choosing All Results prints a report for all groups in the current tabbed page. If the results are not grouped, then all results print for the current tabbed page.
- **Active Group**: If the results are grouped, you can select a group in the current tabbed page, and the printed report contains the results from the selected group.
- **Selected Rows**: You can select single or multiple rows in the audit results report view. Choosing Selected Rows prints a report for such selections.

4. Each tabbed page can contain a list of audits (when the audits are ungrouped) or a group tree with a list of the selected group (when the audits are grouped).

   Note: Unless the results have been grouped using the Group by command, the Active Group option is not enabled in the dialog window.

5. If desired, specify the print zoom factor in the Print zoom field, or check Fit to page if you want to print the results on a single page. If Fit to page is checked, the Print zoom field is disabled.
6. If necessary, adjust the page and printer settings:

- Click the Print list box, and choose the Print dialog box command to select the target printer.
- Choose **Tools** ▶ **Options** and open **Together** ▶ **(level)** ▶ **Diagram** ▶ **Print** options to set up the paper size, orientation, and margins.

  Tip: Click the drop-down arrow to the right of the Preview option to open the preview pane. Use the Preview zoom (auto) slider, or Auto preview zoom check box as required. Click the upward arrow to the right of the Preview option to close the preview pane.

7. Click Print to open the system print dialog box, and send the file to the printer.

See Also

Quality Assurance overview (see page 98)

Viewing audit result (see page 274)

Print Audit dialog box (see page 970)

2.9.11.3 Running Audits

Audits automatically check for conformance to standard or user-defined style, maintenance, and robustness guidelines. Before running audits, make sure that the code being audited is compilable. If your source code contains errors, or some libraries and paths are not included, audits might produce inaccurate results.

Warning: This feature is available for implementation projects only.

To run audits:

1. Open an implementation project.
2. Open the **Model View**.
3. Right-click the project root node. QA Audits on the context menu. The **Audits** dialog window opens.
4. In this dialog window:

- In the Scope list box, choose the code to run the set of audits on.
- Model processes the entire project.
- Selection processes only the specific classes, namespaces, or diagrams currently selected in the Diagram or Model View.

  Tip: If you have not selected any items in the Diagram or Model View, the Scope option defaults to the entire project.

5. If you want to run audits on specific classes, namespaces, or diagrams, make sure you correctly select them before you open the Audits dialog window.
6. Choose the audits to run. As you click an audit, the description for each audit is shown in the lower pane of the dialog box.
7. For each audit, the severity level and other audit-specific options are displayed in the right-hand panel of the Audits dialog box. Change the settings if necessary.
8. When you have selected your set of audits, click Start. The **Operation in progress** dialog box opens displaying a status bar that indicates the progress completed. The status bar will display until the process finishes.

9. If necessary, click Cancel to abort the process.

   Note: Audits run in the command thread, so you cannot edit the project while they are being processed.

The Audits Results Pane opens automatically, displaying the results. In the results table, right-click any line to open the context menu and use its commands to perform operations with the report.

See Also

Quality Assurance facilities overview (see page 98)

Viewing the audit results (see page 274)

2.9.11.4 Viewing Audit Results

When viewing audit results, you can compare and organize items in the results report.

The results report is tightly connected with the diagram elements and the source code. Using the report, you can navigate to the specific location of the violation.

Warning: This feature is available for implementation projects only.

Use the following techniques when viewing audit results:

1. Sort all the items according to the values for a specific column
2. Group items according to the current column
3. Navigate to the specific location of the violation

To sort all the items according to the values for a specific column:

1. Switch to the audit results table.
2. Click the column heading. The arrow in the heading displays whether sorting is ascending or descending.

To group items according to the current column:

1. Right-click the Audit results table and choose Group By. This enables you to organize the results by changing the relationship of rows and columns.
2. To ungroup the results, right-click the table, and choose Ungroup.

To navigate to the specific location of the violation:

1. Select any element in the results report.
2. Choose Open on the context menu (or just double click the row) to navigate directly to the source code.

See Also

Quality Assurance Facilities Overview (see page 98)

Running Audits (see page 273)

2.9.11.5 Working with a Set of Audits

To create a set of audits:

1. On the main menu choose **Tools**▶**Together**▶**QA Audits**. The dialog window **QA Audits** opens.
2. Toolbar buttons in the dialog window provide commands for working with the sets of audits.

3. If you want to base your new saved set on the default set, click the **Set default audit set** button.
4. If you want to base it on a previously created custom set, click the **Load set** button, then choose the desired saved `.adt` file.
5. Go through the individual audits and check those you want to include in the set, or clear those you do not want to include.
6. Select all the items in a group by checking the group name.
7. When you complete your selection, click the **Save set** button, and specify the location and filename for new set file.

To use a saved set of audits:

1. On the main menu choose **Tools ▶ Together ▶ QA Audits**. The dialog window **QA Audits** opens.
2. Click the **Load Set** button and choose the `.adt` file you want to use.
3. Click **Start**.

   Tip: You might want to include the `.adt` files in your backup routine.

See Also

Quality Assurance facilities overview (see page 98)

Viewing the audit results (see page 274)

2.9.11.6 Creating a Metrics Chart

You can create a chart in the **Metric Results Pane**.

Metrics charts are created in temporary files which are deleted when the charts are closed. However, you can save graphical information in text files, export it to the desired graphical format, and include graphics in project.

To create a bar chart:

1. Select a column that contains the result for the desired metric.
2. Right-click and choose Bar Chart.

To create a Kiviat chart:

1. Select the row that contains the results for the desired element.
2. Right-click and choose Kiviat Chart.

To save a chart:

1. Right-click the chart tab and choose Save.
2. In the Save graph dialog box, navigate to the target location and click Save.

To export a chart to image:

1. Select the desired chart.
2. On the main menu, choose File | Export diagram to image.
3. In the Export diagram to image dialog, specify the zoom factor and image dimensions.
4. Click Save.

To add a chart to project:

1. Select the desired chart.
2. On the main menu, choose File | Move [chart name] to Project.
3. On the submenu, select a project within the current project group.

See Also

Quality Assurance facilities overview (↗ see page 98)

Viewing the metric results (↗ see page 276)

2.9.11.7 Running Metrics

Before running metrics, make sure that the code being analyzed can be compiled. If your source code contains errors or some libraries and paths are not included, metrics might produce inaccurate results.

Warning: This feature is available for implementation projects only.

To run metrics:

1. Open an implementation project.
2. Open the **Model View**.
3. Right-click the project root node. QA Metrics on the context menu. The **Metrics** dialog window opens.
4. In this dialog window:

- In Scope, choose what to run metrics on: Model processes the entire project.
- Selection processes only the specific classes, packages, or diagrams currently selected in the diagram or Model View.

5. Choose the metrics you want to analyze. Each metric displays a description in the lower panel of the Metrics dialog box.

   Tip: If nothing is currently selected in the diagram or navigator view, the Selection scope is not available. If you want to run metrics on specific classes, packages, or diagrams, make sure you correctly select them before you open the Metrics dialog window.

6. For each metric there are settings for options such as limits and granularity in the right-hand panel of the Metrics dialog box. Change the settings if necessary.
7. When you have selected your set of metrics, click Start.

   Note: Metrics run in the command thread, so you cannot edit the project while they are being processed.

Result: The Metrics Results Pane opens automatically displaying the results.

See Also

Quality Assurance facilities overview (↗ see page 98)

Viewing the metric results (↗ see page 276)

2.9.11.8 Viewing Metric Results

Use the following techniques when viewing metric results:

1. Sort results by column
2. Filter results
3. Update results
4. Navigate to the source code
5. View the metric description

To sort results by column:

1. Select the desired column in the metrics result table.
2. Click the column header to change the sorting order.

To filter results:

1. You can filter the displayed results to improve the meaningfulness of the results report.
2. Use the following toolbar buttons to show and hide elements:

Button
Namespaces
Classes
Methods
Child elements

To update results:

1. You can update or refresh the results table.
2. Use the following Tool Palette buttons:

Button	Description
Refresh	Recalculate the results that are currently displayed
Restart	Open the Metrics dialog window, define new settings and start new metrics analysis.

To navigate to the source code:

1. Select the row in the results table that is of interest to you
2. Right-click and choose Open on the context menu to navigate directly to it in the source code.

To view the metric description:

1. Select the column in the results table that corresponds to the metrics of interest to you.
2. Right-click and choose Show description on the context menu.

See Also

Quality Assurance facilities overview (see page 98)

Running metrics (see page 276)

2.9.11.9 Working with a Set of Metrics

To create a set of metrics:

1. On the main menu choose **Tools ▶ Together ▶ QA Metrics**. The dialog window **QA Metrics** opens.
2. Toolbar buttons in the dialog window provide commands for working with the sets of metrics.
3. If you want to base your new saved set on the default set, click the **Set default metric set** button.
4. If you want to base it on a previously created custom set, click the **Load set** button, then choose the desired saved `.mts` file.
5. Go through the individual metrics and check those you want to include in the set, or clear those you do not want to include.
6. Select all the items in a group by checking the group name.

7. When you complete your selection, click the **Save set** button, and specify the location and filename for new set file.

To use a saved set of metrics:

1. On the main menu choose **Tools ▶ Together ▶ QA Metrics**. The dialog window **QA Metrics** opens.
2. Click the **Load set** button and choose the `.mts` file you want to use.
3. Click **Start**.

   Tip: You might want to include the `.mts` files in your backup routine.

See Also

Quality Assurance facilities overview (↗ see page 98)

Running metrics (↗ see page 276)

3 Reference

Topics

Name	Description
Delphi Reference (↗ see page 280)	This section describes the Delphi language, Delphi compiler directives, and errors that may arise in Delphi code.
RAD Studio Dialogs and Commands (↗ see page 730)	This section contains help for dialogs and menu commands in the RAD Studio user interface.
Keyboard Mappings (↗ see page 1068)	The following topics list the keyboard mappings available in RAD Studio. Use the **Tools** ▶ **Options** ▶ **Editor Options** ▶ **Key Mappings** page to change the default keyboard mapping.
Command Line Switches and File Extensions (↗ see page 1082)	The following topic lists the IDE command line switches and options.
Together Reference (↗ see page 1086)	This section contains links to the reference material for UML modeling with Together.

3.1 Delphi Reference

This section describes the Delphi language, Delphi compiler directives, and errors that may arise in Delphi code.

Topics

Name	Description
Delphi Compiler Directives (List) (↗ see page 280)	The following topic lists the RAD Studio compiler directives.
Delphi Compiler Errors (↗ see page 311)	The following topics describe the various types of compiler errors and warnings, along with resolutions to many issues you may face while using this product.
Delphi Language Guide (↗ see page 512)	The Delphi Language guide describes the Delphi language as it is used in CodeGear development tools. This book describes the Delphi language on both the Win32, and .NET development platforms. Specific differences in the language between the two platforms are marked as appropriate.

3.1.1 Delphi Compiler Directives (List)

The following topic lists the RAD Studio compiler directives.

Topics

Name	Description
Delphi compiler directives (↗ see page 282)	Each Delphi compiler directive is classified as either a switch, parameter, or conditional compilation directive. A compiler directive is a comment with a special syntax. Compiler directives can be placed wherever comments are allowed. A compiler directive starts with a $ as the first character after the opening comment delimiter, immediately followed by a name (one or more letters) that designates the particular directive. You can include comments after the directive and any necessary parameters. Three types of directives are described in the following topics: • **Switch directives** turn particular compiler features on or off. For the single-letter versions, you add... more (↗ see page 282)
Align fields (Delphi) (↗ see page 283)	
Application type (Delphi) (↗ see page 283)	
Assert directives (Delphi) (↗ see page 284)	
Autoboxing (Delphi for .NET) (↗ see page 284)	
Boolean short-circuit evaluation (Delphi compiler directive) (↗ see page 285)	
Conditional compilation (Delphi) (↗ see page 285)	Conditional compilation is based on the existence and evaluation of constants, the status of compiler switches, and the definition of conditional symbols. Conditional symbols work like Boolean variables: They are either defined (true) or undefined (false). Any valid conditional symbol is treated as false until it has been defined. The $DEFINE directive sets a specified symbol to true, and the $UNDEF directive sets it to false. You can also define a conditional symbol by using the -D switch with the command-line compiler or by adding the symbol to the Conditional Defines box on the Directories/Conditionals page of the Project|Options dialog.... more (↗ see page 285)
Debug information (Delphi) (↗ see page 287)	
DEFINE directive (Delphi) (↗ see page 287)	
DENYPACKAGEUNIT directive (Delphi) (↗ see page 287)	
Description (Delphi) (↗ see page 288)	
DESIGNONLY directive (Delphi) (↗ see page 288)	
ELSE (Delphi) (↗ see page 288)	
ELSEIF (Delphi) (↗ see page 289)	

ENDIF directive (see page 289)	
Executable extension (Delphi) (see page 289)	
Export symbols (Delphi) (see page 290)	
Extended syntax (Delphi) (see page 290)	
External Symbols (Delphi) (see page 290)	
Floating Point Exception Checking (Delphi) (see page 291)	
Hints (Delphi) (see page 291)	
HPP emit (Delphi) (see page 292)	
IFDEF directive (Delphi) (see page 292)	
IF directive (Delphi) (see page 292)	
IFEND directive (Delphi) (see page 293)	
IFNDEF directive (Delphi) (see page 294)	
IFOPT directive (Delphi) (see page 294)	
Image base address (see page 294)	
Implicit Build (Delphi) (see page 295)	
Imported data (see page 295)	
Include file (Delphi) (see page 295)	
Input output checking (Delphi) (see page 296)	
Compiler directives for libraries or shared objects (Delphi) (see page 296)	
Link object file (Delphi) (see page 297)	
Local symbol information (Delphi) (see page 297)	
Long strings (Delphi) (see page 298)	
Memory allocation sizes (Delphi) (see page 298)	
MESSAGE directive (Delphi) (see page 299)	
METHODINFO directive (Delphi) (see page 299)	
Minimum enumeration size (Delphi) (see page 299)	
Open String Parameters (Delphi) (see page 300)	
Optimization (Delphi) (see page 300)	
Overflow checking (Delphi) (see page 301)	
Pentium-safe FDIV operations (Delphi) (see page 301)	
NODEFINE (see page 302)	
NOINCLUDE (Delphi) (see page 302)	
Range checking (see page 302)	
Real48 compatibility (Delphi) (see page 302)	
Regions (Delphi and C#) (see page 303)	
Resource file (Delphi) (see page 303)	
RUNONLY directive (Delphi) (see page 304)	
Runtime type information (Delphi) (see page 304)	
Symbol declaration and cross-reference information (Delphi) (see page 305)	
Type-checked pointers (Delphi) (see page 305)	
UNDEF directive (Delphi) (see page 306)	
Unsafe Code (Delphi for .NET) (see page 306)	
Var-string checking (Delphi) (see page 306)	
Warning messages (Delphi) (see page 307)	
Warnings (Delphi) (see page 308)	
Weak packaging (see page 308)	
Stack frames (Delphi) (see page 309)	
Writeable typed constants (Delphi) (see page 309)	
PE (portable executable) header flags (Delphi) (see page 310)	
Reserved address space for resources (Delphi) (see page 310)	

3.1.1.1 Delphi compiler directives

Each Delphi compiler directive is classified as either a switch, parameter, or conditional compilation directive.

A compiler directive is a comment with a special syntax. Compiler directives can be placed wherever comments are allowed. A compiler directive starts with a $ as the first character after the opening comment delimiter, immediately followed by a name (one or more letters) that designates the particular directive. You can include comments after the directive and any necessary parameters.

Three types of directives are described in the following topics:

- **Switch directives** turn particular compiler features on or off. For the single-letter versions, you add either + or - immediately after the directive letter. For the long version, you supply the word "on" or "off."

* Switch directives are either global or local.

- Global directives affect the entire compilation and must appear before the declaration part of the program or the unit being compiled.
- Local directives affect only the part of the compilation that extends from the directive until the next occurrence of the same directive. They can appear anywhere.

Switch directives can be grouped in a single compiler directive comment by separating them with commas with no intervening spaces. For example:

```
{$B+,R-,S-}
```

- **Parameter directives.** These directives specify parameters that affect the compilation, such as file names and memory sizes.
- **Conditional directives.** These directives cause sections of code to be compiled or suppressed based on specified conditions, such as user-defined conditional symbols.

All directives, except switch directives, must have at least one space between the directive name and the parameters. Here are some examples of compiler directives:

```
{$B+}
    {$STACKCHECKS ON}
    {$R- Turn off range checking}
    {$I TYPES.INC}
    {$M 32768,4096}
    {$DEFINE Debug}
    {$IFDEF Debug}
    {$ENDIF}
```

You can insert compiler directives directly into your source code. You can also change the default directives for the command-line compiler, dccil and the IDE, bds.exe.

The Project|Options dialog box contains many of the compiler directives; any changes you make to the settings there will affect all units whenever their source code is recompiled in subsequent compilations of that project. If you change a compiler switch and compile, none of your units will reflect the change; but if you Build All, all units for which you have source code will be recompiled with the new settings.

When using the command-line compiler, you can specify compiler directives on the command line; for example,

```
DCCIL -$R+ MYPROG
```

If you are working in the Code editor and want a quick way to see what compiler directives are in effect, press Ctrl+O O. You will see the current settings in the edit window at the top of your file.

See Also

List of Compiler Directives (↗ see page 280)

Conditional Compilation

3.1.1.2 Align fields (Delphi)

Type	Switch
Syntax	{$A+}, {$A-}, {$A1}, {$A2}, {$A4}, or {$A8} {$ALIGN ON}, {$ALIGN OFF}, {$ALIGN 1}, {$ALIGN 2}, {$ALIGN 4}, or {$ALIGN 8}
Default	{$A8} {$ALIGN 8}
Scope	Local

Remarks

The **$A** directive controls alignment of fields in Delphi record types and class structures.

In the **{$A1}** or **{$A-}** state, fields are never aligned. All record and class structures are packed.

In the **{$A2}** state, fields in record types that are declared without the **packed** modifier and fields in class structures are aligned on word boundaries.

In the **{$A4}** state, fields in record types that are declared without the **packed** modifier and fields in class structures are aligned on double-word boundaries.

In the **{$A8}** or **{$A+}** state, fields in record types that are declared without the **packed** modifier and fields in class structures are aligned on quad word boundaries.

Record type field alignment is described in the Delphi Language Guide.

Regardless of the state of the **$A** directive, variables and typed constants are always aligned for optimal access. In the **{$A8}** state, execution will be faster.

See Also

Record types (see page 566)

3.1.1.3 Application type (Delphi)

Type	Parameter
Syntax	{$APPTYPE GUI} or {$APPTYPE CONSOLE}
Default	{$APPTYPE GUI}
Scope	Global

Remarks

This directive is used in Delphi Windows programming only.

The $APPTYPE directive controls whether to generate a Win32 console or graphical user interface application.

In the {$APPTYPE GUI} state, the compiler generates a graphical user interface application. This is the normal state for a Delphi application.

In the {$APPTYPE CONSOLE} state (equivalent to the /CC command-line option), the compiler generates a console application. When a console application is started, the Input and Output standard text files are automatically associated with the console window.

Setting {$APPTYPE CONSOLE} can be convenient for debugging as it allows you to use WriteLn statements in your program without having to explicitly open an output file.

The IsConsole Boolean variable declared in the System unit can be used to detect whether a program is running as a console or graphical user interface application.

The $APPTYPE directive is meaningful only in a program. It should not be used in a library, unit, or package.

3.1.1.4 Assert directives (Delphi)

Type	Switch
Syntax	{$C+} or {$C-} {$ASSERTIONS ON} or {$ASSERTIONS OFF}
Default	{$C+} {$ASSERTIONS ON}
Scope	Local

Remarks

The **$C** directive enables or disables the generation of code for assertions in a Delphi source file. {**$C+**} is the default.

Since assertions are not usually used at runtime in shipping versions of a product, compiler directives that disable the generation of code for assertions are provided. {**$C-**} will disable assertions.

3.1.1.5 Autoboxing (Delphi for .NET)

Type	Switch
Syntax	{$AUTOBOX ON}, {$AUTOBOX OFF}
Default	{$AUTOBOX OFF}
Scope	Local

Remarks

The **$AUTOBOX** directive controls whether value types are automatically “boxed” into reference types.

The following code will not compile by default; the compiler halts with a message that `I` and `Obj` have incompatible types.

```
var
    I: Integer;
    Obj: TObject;
begin
    I:=5;
    Obj:=I; // compilation error
end.
```

Inserting **{$AUTOBOX ON}** anywhere before the offending line will remove the error, so this code compiles:

```
var
    I: Integer;
    Obj: TObject;
begin
    I:=5;
    {$AUTOBOX ON}
    Obj:=I; // I is autoboxed into a TObject
end.
```

Reference types can not be automatically “unboxed” into value types, so a typecast is required to turn the TObject into an Integer:

```
var
    I: Integer;
    Obj: TObject;
begin
    I:=5;
    {$AUTOBOX ON}
    Obj:=I; // OK
    // I:=Obj; // Can't automatically unbox; compilation error
    I:=Integer(Obj); // this works
end.
```

Turning on autoboxing can be convenient, but it makes Delphi less type safe, so it can be dangerous. With autoboxing, some errors that would otherwise be caught during compilation may cause problems at runtime. Boxing values into object references also consumes additional memory and degrades execution performance. With **{$AUTOBOX ON}**, you run the risk of not realizing how much of this data conversion is happening silently in your code. **{$AUTOBOX OFF}** is recommended for improved type checking and faster runtime execution.

The **$AUTOBOX** directive has no effect in Delphi for Win32.

3.1.1.6 Boolean short-circuit evaluation (Delphi compiler directive)

Type	Switch
Syntax	{$B+} or {$B-} {$BOOLEVAL ON} or {$BOOLEVAL OFF}
Default	{$B-} {$BOOLEVAL OFF}
Scope	Local

Remarks

The $B directive switches between the two different models of Delphi code generation for the **and** and **or** Boolean operators.

In the {$B+} state, the compiler generates code for complete Boolean expression evaluation. This means that every operand of a Boolean expression built from the and and or operators is guaranteed to be evaluated, even when the result of the entire expression is already known.

In the {$B-} state, the compiler generates code for short-circuit Boolean expression evaluation, which means that evaluation stops as soon as the result of the entire expression becomes evident in left to right order of evaluation.

See Also

Boolean operators (↗ see page 720)

3.1.1.7 Conditional compilation (Delphi)

Conditional compilation is based on the existence and evaluation of constants, the status of compiler switches, and the definition of conditional symbols.

Conditional symbols work like Boolean variables: They are either defined (true) or undefined (false). Any valid conditional symbol is treated as false until it has been defined. The $DEFINE directive sets a specified symbol to true, and the $UNDEF directive sets it to false. You can also define a conditional symbol by using the -D switch with the command-line compiler or by adding the symbol to the Conditional Defines box on the Directories/Conditionals page of the Project|Options dialog.

The conditional directives $IFDEF, $IFNDEF, $IF, $ELSEIF, $ELSE, $ENDIF, and $IFEND allow you to compile or suppress code based on the status of a conditional symbol. $IF and $ELSEIF allow you to base conditional compilation on declared Delphi identifiers. $IFOPT compiles or suppresses code depending on whether a specified compiler switch is enabled.

For example,

```
{$DEFINE DEBUG}
    {$IFDEF DEBUG}
    Writeln('Debug is on.');  // this code executes
    {$ELSE}
    Writeln('Debug is off.');  // this code does not execute
    {$ENDIF}
    {$UNDEF DEBUG}
    {$IFNDEF DEBUG}
    Writeln('Debug is off.');  // this code executes
    {$ENDIF}
```

Conditional-directive constructions can be nested up to 32 levels deep. For every {$IFxxx}, the corresponding {$ENDIF} or {$IFEND} must be found within the same source file.

Conditional symbols must start with a letter, followed by any combination of letters, digits, and underscores; they can be of any length, but only the first 255 characters are significant. The following standard conditional symbols are defined:

VER<nnn> Always defined, indicating the version number of the Delphi compiler. (Each compiler version has a corresponding predefined symbol. For example, compiler version 18.0 has VER180 defined.)

MSWINDOWS Indicates that the operating environment is Windows. Use MSWINDOWS to test for any flavor of the Windows platform instead of WIN32.

WIN32 Indicates that the operating environment is the Win32 API. Use WIN32 for distinguishing between specific Windows platforms, such as 32-bit versus 64-bit Windows. In general, don't limit code to WIN32 unless you know for sure that the code will not work in WIN64. Use MSWINDOWS instead.

CLR Indicates the code will be compiled for the .NET platform.

CPU386 Indicates that the CPU is an Intel 386 or better.

CONSOLE Defined if an application is being compiled as a console application.

CONDITIONALEXPRESSIONS Tests for the use of $IF directives.

For example, to find out the version of the compiler and run-time library that was used to compile your code, you can use $IF with the CompilerVersion, RTLVersion and other constants:

```
{$IFDEF CONDITIONALEXPRESSIONS}
   {$IF CompilerVersion >= 17.0}
     {$DEFINE HAS_INLINE}
   {$IFEND}
   {$IF RTLVersion >= 14.0}
     {$DEFINE HAS_ERROUTPUT}
   {$IFEND}
{$ENDIF}
```

Note: Conditional symbols are not Delphi identifiers and cannot be referenced in actual program code. Similarly, Delphi identifiers cannot be referenced in any conditional directives other than $IF and $ELSEIF.

Note: Conditional definitions are evaluated only when source code is recompiled. If you change a conditional symbol's status and then rebuild a project, source code in unchanged units may not be recompiled. Use Project|Build All Projects to ensure everything in your project reflects the current status of conditional symbols.

See Also

Delphi Compiler Directives (List) (see page 280)

3.1.1.8 Debug information (Delphi)

Type	Switch
Syntax	{$D+} or {$D-} {$DEBUGINFO ON} or {$DEBUGINFO OFF}
Default	{$D+} {$DEBUGINFO ON}
Scope	Global

Remarks

The $D directive enables or disables the generation of debug information. This information consists of a line-number table for each procedure, which maps object-code addresses into source text line numbers.

For units, the debug information is recorded in the unit file along with the unit's object code. Debug information increases the size of unit file and takes up additional memory when compiling programs that use the unit, but it does not affect the size or speed of the executable program.

When a program or unit is compiled in the {$D+} state, the integrated debugger lets you single-step and set breakpoints in that module.

The Include debug info (Project|Options|Linker) and Map file (Project|Options|Linker) options produce complete line information for a given module only if you've compiled that module in the {$D+} state.

The $D switch is usually used in conjunction with the $L switch, which enables and disables the generation of local symbol information for debugging.

3.1.1.9 DEFINE directive (Delphi)

Type	Conditional compilation
Syntax	{$DEFINE name}

Remarks

Defines a Delphi conditional symbol with the given name. The symbol is recognized for the remainder of the compilation of the current module in which the symbol is declared, or until it appears in an {$UNDEF name} directive. The {$DEFINE name} directive has no effect if name is already defined.

3.1.1.10 DENYPACKAGEUNIT directive (Delphi)

Type	Switch
Syntax	{$DENYPACKAGEUNIT ON} or {$DENYPACKAGEUNIT OFF}
Default	{$DENYPACKAGEUNIT OFF}
Scope	Local

Remarks

The **{$DENYPACKAGEUNIT ON}** directive prevents the Delphi unit in which it appears from being placed in a package.

3.1.1.11 Description (Delphi)

Type	Parameter
Syntax	{$DESCRIPTION 'text'}
Scope	Global

Remarks

The $D directive inserts the text you specify into the module description entry in the header of an executable, DLL, or package. Traditionally the text is a name, version number, and copyright notice, but you may specify any text of your choosing. For example:

```
{$D 'My Application version 12.5'}
```

The string can't be longer than 256 bytes. The description is usually not visible to end users. To mark you executable files with descriptive text, version and copyright information for the benefit of end users, use version info resources.

Note: The text description must be included in quotes.

3.1.1.12 DESIGNONLY directive (Delphi)

Type	Switch
Syntax	{$DESIGNONLY ON} or {$DESIGNONLY OFF}
Default	{$DESIGNONLY OFF}
Scope	Local

Remarks

The **{DESIGNONLY ON}** directive causes the package where it occurs to be compiled for installation in the IDE.

Place the **DESIGNONLY** directive only in .dpk files.

See Also

Package-specific compiler directives (↗ see page 640)

3.1.1.13 ELSE (Delphi)

Type	Conditional compilation
Syntax	{$ELSE}

Remarks

Switches between compiling and ignoring the source code delimited by the previous {$IFxxx} and the next {$ENDIF} or {$IFEND}.

3.1.1.14 ELSEIF (Delphi)

Type	Conditional compilation
Syntax	{$ELSEIF}

Remarks

The **$ELSEIF** directive allows multi-part conditional blocks where at most one of the conditional blocks will be taken. **$ELSEIF** is a combination of a **$ELSE** and a **$IF**.

For example:

```
{$IFDEF  foobar}
       do_foobar
    {$ELSEIF  RTLVersion >= 14}
        blah
    {$ELSEIF  somestring = 'yes'}
        beep
    {$ELSE}
        last chance
    {$IFEND}
```

Of these four cases, only one will be taken. If none of the first three conditions is true, then the **$ELSE** clause will be taken. **$ELSEIF** must be terminated by **$IFEND**. **$ELSEIF** cannot appear after **$ELSE**. Conditions are evaluated top to bottom like a normal "if ... else if ... else " sequence. In the example above, if foobar is not defined, RTLVersion is 15, and somestring = 'yes', only the "blah" block will be taken not the "beep" block, even though the conditions for both are true.

3.1.1.15 ENDIF directive

Type	Conditional compilation
Syntax	{$ENDIF}

Remarks

Ends the conditional compilation initiated by the last {$IFxxx} directive.

Note: $IF and $ELSEIF directives terminate with $IFEND rather than $ENDIF.

3.1.1.16 Executable extension (Delphi)

Type	Parameter
Syntax	{$E extension} {$EXTENSION extension}

The **$E** directive sets the extension of the executable file generated by the compiler. It is often used with the resource-only DLL mechanism.

For example, placing **{$E de**u**}** in a library module produces a DLL with a .deu extension: filename.deu. If you create a library module that simply references German forms and strings, you could use this directive to produce a DLL with the .deu extension. The startup code in the runtime library looks for a DLL whose extension matches the locale of the system—for German settings, it looks for .deu—and loads resources from that DLL.

3.1.1.17 Export symbols (Delphi)

Type	Switch
Syntax	{$ObjExportAll On} or {$ObjExportAll Off}
Default	{$ObjExportAll Off}
Scope	Global

The **{$ObjExportAll On}** directive exports all symbols in the unit file in which it occurs. This allows the C++ compiler to create packages containing Delphi-generated object files.

3.1.1.18 Extended syntax (Delphi)

Type	Switch
Syntax	{$X+} or {$X-} {$EXTENDEDSYNTAX ON} or {$EXTENDEDSYNTAX OFF}
Default	{$X+} {$EXTENDEDSYNTAX ON}
Scope	Global

Remarks

Note: Note: The $X directive is provided for backward compatibility. You should not use the {$X-} mode when writing Delphi applications.

The $X directive enables or disables Delphi's extended syntax:

- Function statements. In the {$X+} mode, function calls can be used as procedure calls; that is, the result of a function call can be discarded, rather than passed to another function or used in an operation or assignment. Generally, the computations performed by a function are represented through its result, so discarding the result makes little sense. Sometimes, however, a function is called because it performs a task such as setting the value of a global variable, without producing a useful result.
- The Result variable. In the {$X+} mode, the predefined variable Result can be used within a function body to hold the function's return value.
- Null-terminated strings. In the {$X+} mode, Delphi strings can be assigned to zero-based character arrays (**array**[0..X] **of** Char), which are compatible with PChar types.

See Also

Function declarations (↗ see page 662)

Working with null-terminated strings (↗ see page 561)

3.1.1.19 External Symbols (Delphi)

Type	Parameter
Syntax	{$EXTERNALSYM identifier}

The EXTERNALSYM directive prevents the specified Delphi symbol from appearing in header files generated for C++. If an overloaded routine is specified, all versions of the routine are excluded from the header file.

3.1.1.20 Floating Point Exception Checking (Delphi)

Type	Switch
Syntax	{$FINITEFLOAT ON}, {$FINITEFLOAT OFF}
Default	{$FINITEFLOAT ON}
Scope	Global

Remarks

The **$FINITEFLOAT** directive controls the handling of floating point overflow and underflow, and invalid floating point operations such as division by zero.

In the **{$FINITEFLOAT ON}** state, which is the default, the results of floating point calculations are checked, and an exception is raised when there is an overflow, underflow, or invalid operation. In the **{$FINITEFLOAT OFF}** state, such floating point calculations will return NAN, -INF, or +INF.

Extra runtime processing is required to check the results of floating point calculations and raise exceptions. If your Delphi code uses floating point operations but does not require strict enforcement of overflow/underflow exceptions, you can turn **{$FINITEFLOAT OFF}** to get slightly faster runtime execution.

Note: Most code in .NET runs without floating point checks. However, Delphi has traditionally provided strict floating point semantics. If you have Delphi code that relies on exceptions to be raised in overflow and underflow conditions, you should retain the default setting (**{$FINITEFLOAT ON}**).

See Also

Internal Data Formats (↗ see page 645)

3.1.1.21 Hints (Delphi)

Type	Switch
Syntax	{$HINTS ON} or {$HINTS OFF}
Default	{$HINTS ON}
Scope	Local

Remarks

The **$HINTS** directive controls the generation of hint messages by the Delphi compiler.

In the {**$HINTS ON**} state, the compiler issues hint messages when detecting unused variables, unused assignments, **for** or **while** loops that never execute, and so on. In the {**$HINTS OFF**} state, the compiler generates no hint messages.

By placing code between {**$HINTS OFF**} and {**$HINTS ON**} directives, you can selectively turn off hints that you don't care about. For example,

```
{$HINTS OFF}
    procedure Test;
    var
     I: Integer;
    begin
    end;
    {$HINTS ON}
```

Because of the **$HINTS** directives the compiler will not generate an unused variable hint when compiling the procedure above.

3.1.1.22 HPP emit (Delphi)

Type	Parameter
Syntax	{$HPPEMIT 'string'}

The HPPEMIT directive adds a specified symbol to the header file generated for C++. Example: {$HPPEMIT 'typedef double Weight' }.

HPPEMIT directives are output into the "user supplied" section at the top of the header file in the order in which they appear in the Delphi file.

3.1.1.23 IFDEF directive (Delphi)

Type	Conditional compilation
Syntax	{$IFDEF name}

Remarks

Compiles the Delphi source code that follows it if name is defined.

3.1.1.24 IF directive (Delphi)

Type	Conditional compilation
Syntax	{$IF expression}

Remarks

Compiles the Delphi source code that follows it if expression is true. expression must conform to Delphi syntax and return a Boolean value; it may contain declared constants, constant expressions, and the functions Defined and Declared.

For example,

```
{$DEFINE CLX}
    const LibVersion = 2.1;
    {$IF Defined(CLX) and (LibVersion > 2.0) }
      ...  // this code executes
    {$ELSE}
      ...  // this code doesn't execute
    {$IFEND}
    {$IF Defined(CLX) }
      ...  // this code executes
    {$ELSEIF LibVersion > 2.0}
      ...  // this code doesn't execute
    {$ELSEIF LibVersion = 2.0}
      ...  // this code doesn't execute
    {$ELSE}
      ...  // this code doesn't execute
    {$IFEND}
    {$IF Declared(Test)}
      ... // successful
    {$IFEND}
```

The special functions Defined and Declared are available only within $IF and $ELSEIF blocks. Defined returns true if the argument passed to it is a defined conditional symbol. Declared returns true if the argument passed to it is a valid declared Delphi identifier visible within the current scope.

If the identifiers referenced in the conditional expression do not exist, the conditional expression will be evaluated as false:

```
{$IF NoSuchVariable > 5}
     WriteLn('This line doesn''t compile');
    {$IFEND}
```

The $IF and $ELSEIF directives are terminated with $IFEND, unlike other conditional directives that use the $ENDIF terminator. This allows you to hide $IF blocks from earlier versions of the compiler (which do not support $IF or $ELSEIF) by nesting them within old-style $IFDEF blocks. For example, the following construction would not cause a compilation error:

```
     {$UNDEF NewEdition}
         {$IFDEF NewEdition}
           {$IF LibVersion > 2.0}
             ...
           {$IFEND}
         {$ENDIF}
```

$IF supports evaluation of typed constants, but the compiler doesn't allow typed constants within constant expressions. As a result,

```
const Test: Integer = 5;
    {$IF SizeOf(Test) > 2}
    ...
```

is valid, while

```
const Test: Integer = 5;
    {$IF Test > 2 }         // error
    ...
```

generates a compilation error.

If your code needs to be portable between various versions of Delphi, or platforms (such as .NET), you will need to test whether or not this directive is supported by the compiler. You can surround your code with the following directives:

```
$IFDEF conditionalexpressions
    .          // code including IF directive
    .          // only executes if supported
    $ENDIF
```

Note: To test if code is being compiled on the .NET platform, use the identifier `CLR`, for example, `{$IF NOT DEFINED(CLR)}`.

3.1.1.25 IFEND directive (Delphi)

Type	Conditional compilation
Syntax	{$IFEND}

Remarks

The **$IFEND** directive terminates **$IF** and **$ELSEIF**. This allows **$IF/$IFEND** blocks to be hidden from older compilers inside of **$IFDEF/$ENDIF**, since the older compilers won't recognize **$IFEND** as a directive. **$IF** can only be terminated with **$IFEND**. The **$IFDEF**, **$IFNDEF**, **$IFOPT** directives can only be terminated with **$ENDIF**.

Note: Note: When hiding **$IF** inside **$IFDEF/$ENDIF**, do not use **$ELSE** with the **$IF**. Previous version compilers will interpret the **$ELSE** as part of the **$IFDEF**, producing a compiler error. You can use an {**$ELSEIF** True} as a substitute for {**$ELSE**} in this situation, since the **$ELSEIF** won't be taken if the **$IF** is taken first, and the older compilers will not interpret the **$ELSEIF**. Hiding **$IF** for backwards compatibility is primarily an issue for third party vendors and application developers who need their code to work on a variety of Delphi versions and platforms.

3.1.1.26 IFNDEF directive (Delphi)

Type	Conditional compilation
Syntax	{$IFNDEF name}

Remarks

Compiles the Delphi source code that follows it if name is not defined.

3.1.1.27 IFOPT directive (Delphi)

Type	Conditional compilation
Syntax	{$IFOPT switch}

Remarks

Compiles the Delphi source code that follows it if switch is currently in the specified state. switch consists of the name of a switch option, followed by a + or a - symbol. For example,

```
{$IFOPT R+}
     Writeln('Compiled with range-checking');
    {$ENDIF}
```

compiles the Writeln statement if the $R option is currently active.

3.1.1.28 Image base address

Type	Parameter
Syntax	{$IMAGEBASE number}
Default	{$IMAGEBASE $00400000}
Scope	Global

The $IMAGEBASE directive controls the default load address for an application, DLL, or package. The number argument must be a 32-bit integer value that specifies image base address. The number argument must be greater than or equal to $00010000, and the lower 16 bits of the argument are ignored and should be zero. The number must be a multiple of 64K (that is, a hex number must have zeros as the last 4 digits) otherwise it will be rounded down to the nearest multiple, and you will receive a compiler message.

When a module (application or library) is loaded into the address space of a process, Windows will attempt to place the module at its default image base address. If that does not succeed, that is if the given address range is already reserved by another module, the module is relocated to an address determined at runtime by Windows.

There is seldom, if ever, any reason to change the image base address of an application. For a library, however, it is recommended that you use the $IMAGEBASE directive to specify a non-default image base address, since the default image base address of $00400000 will almost certainly never be available. The recommended address range of DLL images is $40000000 to $7FFFFFFF. Addresses in this range are always available to a process in both Windows NT/2000 and Windows 95/98.

When Windows succeeds in loading a DLL (or package) at its image base address, the load time is decreased because relocation fix-ups do not have to be applied. Furthermore, when the given address range is available in multiple processes that use the library, code portions of the DLL's image can be shared among the processes, thus reducing load time and memory consumption.

Note: Note: The $IMAGEBASE directive overrides any value supplied with the -K command line compiler directive option.

3.1.1.29 Implicit Build (Delphi)

Type	Switch
Syntax	{$IMPLICITBUILD ON} or {$IMPLICITBUILD OFF}
Default	{$IMPLICITBUILD ON}
Scope	Global

Remarks

The **{$IMPLICITBUILD OFF}** directive, intended only for packages, prevents the source file in which it occurs from being implicitly recompiled later. Use **{$IMPLICITBUILD OFF}** in .dpk files when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed. Use of **{$IMPLICITBUILD OFF}** in unit source files is not recommended.

3.1.1.30 Imported data

Type	Switch
Syntax	{$G+} or {$G-} {$IMPORTEDDATA ON} or {$IMPORTEDDATA OFF}
Default	{$G+} {$IMPORTEDDATA ON}
Scope	Local

Remarks

The **{$G-}** directive disables creation of imported data references. Using **{$G-}** increases memory-access efficiency, but prevents a packaged unit where it occurs from referencing variables in other packages.

3.1.1.31 Include file (Delphi)

Type	Parameter
Syntax	{$I filename} {$INCLUDE filename}
Scope	Local

Remarks

The $I parameter directive instructs the compiler to include the named file in the compilation. In effect, the file is inserted in the compiled text right after the {$I filename} directive. The default extension for filename is .pas. If filename does not specify a directory path, then, in addition to searching for the file in the same directory as the current module, Delphi searches in the directories specified in the Search path input box on the Directories/Conditionals page of the Project|Options dialog box (or in the directories specified in a -I option on the `dccil` command line).

To specify a filename that includes a space, surround the file name with single quotation marks: {$I 'My file'}.

There is one restriction to the use of include files: An include file can't be specified in the middle of a statement part. In fact, all statements between the begin and end of a statement part must exist in the same source file.

3.1.1.32 Input output checking (Delphi)

Type	Switch
Syntax	{$I+} or {$I-} {$IOCHECKS ON} or {$IOCHECKS OFF}
Default	{$I+} {$IOCHECKS ON}
Scope	Local

Remarks

The $I switch directive enables or disables the automatic code generation that checks the result of a call to an I/O procedure. I/O procedures are described in the Delphi Language Guide. If an I/O procedure returns a nonzero I/O result when this switch is on, an EInOutError exception is raised (or the program is terminated if exception handling is not enabled). When this switch is off, you must check for I/O errors by calling IOResult.

See Also

Standard routines and I/O (see page 692)

3.1.1.33 Compiler directives for libraries or shared objects (Delphi)

Type	**Parameter**
Syntax	$LIBPREFIX 'string' $LIBSUFFIX 'string' $LIBVERSION 'string'
Defaults	$LIBPREFIX 'lib' or $SOPREFIX 'bpl' $LIBSUFFIX ' ' $LIBVERSION ' '
Scope	Global

Remarks

$LIBPREFIX overrides the default 'lib' or 'bpl' prefix in the output file name. For example, you could specify

{$LIBPREFIX 'dcl'}

for a design-time package, or use the following directive to eliminate the prefix entirely:

{$LIBPREFIX ' '}

$LIBSUFFIX adds a specified suffix to the output file name before the extension.

For example, use

{$LIBSUFFIX '-2.1.3'}

in something.pas to generate

something-2.1.3.dll

$LIBVERSION adds a second extension to the output file name after the extension. For example, use

{$LIBVERSION '-2.1.3'}

in something.pas to generate

libsomething.dll.2.1.3

3.1.1.34 Link object file (Delphi)

Type	Parameter
Syntax	{$L filename} {$LINK filename}
Scope	Local

Remarks

The $L parameter instructs the compiler to link the named file with the program or unit being compiled. The $L directive is used to link with code written in other languages for procedures and functions declared to be external. The named file must be an Intel relocatable object file (.OBJ file). The default extension for filename is .OBJ. If filename does not specify a directory path, then, in addition to searching for the file in the same directory as the current module, Delphi searches in the directories specified in the Search path input box on the Directories/Conditionals page of the Project|Options dialog box (or in the directories specified in the -O option on the dccil command line).

To specify a file name that includes a space, surround the file name with single quotation marks: {$L 'My file'}.

For further details about linking with assembly language, see online Help.

3.1.1.35 Local symbol information (Delphi)

Type	Switch
Syntax	{$L+} or {$L-} {$LOCALSYMBOLS ON} or {$LOCALSYMBOLS OFF}
Default	{$L+} {$LOCALSYMBOLS ON}
Scope	Global

Remarks

The $L switch directive enables or disables the generation of local symbol information. Local symbol information consists of the names and types of all local variables and constants in a module, that is, the symbols in the module's implementation part and the symbols within the module's procedures and functions.

For units, the local symbol information is recorded in the unit file along with the unit's object code. Local symbol information increases the size of unit files and takes up additional memory when compiling programs that use the unit, but it does not affect the size or speed of the executable program.

When a program or unit is compiled in the {$L+} state, the integrated debugger lets you examine and modify the module's local variables. Furthermore, calls to the module's procedures and functions can be examined via the View|Call Stack.

The $L switch is usually used in conjunction with the $D switch, which enables and disables the generation of line-number tables for debugging. The $L directive is ignored if the compiler is in the {$D-} state.

3.1.1.36 Long strings (Delphi)

Type	Switch
Syntax	{$H+} or {$H-} {$LONGSTRINGS ON} or {$LONGSTRINGS OFF}
Default	{$H+} {$LONGSTRINGS ON}
Scope	Local

Remarks

The **$H** directive controls the meaning of the reserved word **string** when used alone in a type declaration. The generic type **string** can represent either a long, dynamically-allocated string (the fundamental type AnsiString) or a short, statically allocated string (the fundamental type ShortString).

By default **{$H+}**, Delphi defines the generic string type to be the long AnsiString. All components in the component libraries are compiled in this state. If you write components, they should also use long strings, as should any code that receives data from component library string-type properties.

The {**$H-**} state is mostly useful for using code from versions of Delphi that used short strings by default. You can locally override the meaning of string-type definitions to ensure generation of short strings. You can also change declarations of short string types to **string**[255] or ShortString, which are unambiguous and independent of the **$H** setting.

3.1.1.37 Memory allocation sizes (Delphi)

Type	Parameter
Syntax	{$M minstacksize,maxstacksize} {$MINSTACKSIZE number} {$MAXSTACKSIZE number}
Default	{$M 16384,1048576}
Scope	Global

Remarks

The **$MINSTACKSIZE** and **$MAXSTACKSIZE** directives are used in Win32 programming only.

The $M directive specifies an application's stack allocation parameters. minstacksize must be an integer number between 1024 and 2147483647 that specifies the minimum size of an application's stack, and maxstacksize must be an integer number between minstacksize and 2147483647 that specifies the maximum size of an application's stack.

If there is not enough memory available to satisfy an application's minimum stack requirement, Windows will report an error upon attempting to start the application.

An application's stack is never allowed to grow larger than the maximum stack size. Any attempt to grow the stack beyond the maximum stack size causes an EStackOverflow exception to be raised.

The $MINSTACKSIZE and $MAXSTACKSIZE directives allow the minimum and maximum stack sizes to be specified separately.

The memory allocation directives are meaningful only in a program. They should not be used in a library or a unit.

For portability considerations between Windows and Linux, you should use the long forms of these directives instead of $M.

3.1.1.38 MESSAGE directive (Delphi)

Syntax	{$MESSAGE HINT|WARN|ERROR|FATAL 'text string' }

Remarks

The Delphi message directive allows source code to emit hints, warnings, and errors just as the compiler does. This is similar to #emit or pragma warn in C and C++.

The message type (HINT, WARN, ERROR, or FATAL) is optional. If no message type is indicated, the default is HINT. The text string is required and must be enclosed in single quotes.

Examples:

```
{$MESSAGE 'Boo!'}                   emits a hint
    {$Message Hint 'Feed the cats'}     emits a hint
    {$messaGe Warn 'Looks like rain.'}  emits a warning
    {$Message Error 'Not implemented'}  emits an error, continues compiling
    {$Message Fatal 'Bang.  Yer dead.'} emits an error, terminates compiler
```

3.1.1.39 METHODINFO directive (Delphi)

Type	Switch
Syntax	{$METHODINFO ON} or {$METHODINFO OFF}
Default	{$METHODINFO OFF}
Scope	Local

The `$METHODINFO` switch directive is only effective when runtime type information (RTTI) has been turned on with the `{$TYPEINFO ON}` switch. In the `{$TYPEINFO ON}` state, the `$METHODINFO` directive controls the generation of more detailed method descriptors in the RTTI for methods in an interface. Though `{$TYPEINFO ON}` will cause some RTTI to be generated for published methods, the level of information is limited. The `$METHODINFO` directive generates much more detailed (and much larger) RTTI for methods, which describes how the parameters of the method should be passed on the stack and/or in registers.

There is seldom, if ever, any need for an application to directly use the `$METHODINFO` compiler switch. The method information adds considerable size to the executable file, and is not recommended for general use.

Note: The Delphi compiler's Win32 web service support code uses method information descriptors in order to pass parameters received in a network packet to the target method. `{$METHODINFO ON}` is used only for web service interface types.

See Also

Runtime type information ($TYPEINFO directive) (↗ see page 304)

Understanding Invokable Interfaces

3.1.1.40 Minimum enumeration size (Delphi)

Type	Parameter
Syntax	{$Z1} or {$Z2} or {$Z4} {$MINENUMSIZE 1} or {$MINENUMSIZE 2} or {$MINENUMSIZE 4}

Default	{$Z1} {$MINENUMSIZE 1}
Scope	Local

The **$Z** directive controls the minimum storage size of Delphi enumerated types.

An enumerated type is stored as an unsigned byte if the enumeration has no more than 256 values, and if the type was declared in the {**$Z1**} state (the default). If an enumerated type has more than 256 values, or if the type was declared in the {**$Z2**} state, it is stored as an unsigned word. Finally, if an enumerated type is declared in the {**$Z4**} state, it is stored as an unsigned double word.

The {**$Z2**} and {**$Z4**} states are useful for interfacing with C and C++ libraries, which usually represent enumerated types as words or double words.

Note: Note: For backwards compatibility with early versions of Delphi and CodeGear Pascal, the directives {**$Z-**} and {**$Z+**} are also supported. They correspond to {**$Z1**} and {**$Z4**}, respectively.

3.1.1.41 Open String Parameters (Delphi)

Type	Switch
Syntax	{$P+} or {$P-} {$OPENSTRINGS ON} or {$OPENSTRINGS OFF}
Default	{$P+} {$OPENSTRINGS ON}
Scope	Local

Remarks

The **$P** directive is meaningful only for code compiled in the {**$H-**} state, and is provided for backwards compatibility with early versions of Delphi and CodeGear Pascal. **$P** controls the meaning of variable parameters declared using the string keyword in the {**$H-**} state. In the {**$P**-} state, variable parameters declared using the string keyword are normal variable parameters, but in the {**$P**+} state, they are open string parameters. Regardless of the setting of the **$P** directive, the openstring identifier can always be used to declare open string parameters.

3.1.1.42 Optimization (Delphi)

Type	Switch
Syntax	{$O+} or {$O-} {$OPTIMIZATION ON} or {$OPTIMIZATION OFF}
Default	{$O+} {$OPTIMIZATION ON}
Scope	Local

The **$O** directive controls code optimization. In the {**$O+**} state, the compiler performs a number of code optimizations, such as placing variables in CPU registers, eliminating common subexpressions, and generating induction variables. In the {**$O-**} state, all such optimizations are disabled.

Other than for certain debugging situations, you should never have a need to turn optimizations off. All optimizations performed by the Delphi compiler are guaranteed not to alter the meaning of a program. In other words, the compiler performs no "unsafe" optimizations that require special awareness by the programmer.

Note: Note: The $O directive can only turn optimization on or off for an entire procedure or function. You can't turn optimization on or off for a single line or group of lines within a routine.

3.1.1.43 Overflow checking (Delphi)

Type	Switch
Syntax	{$Q+} or {$Q-} {$OVERFLOWCHECKS ON} or {$OVERFLOWCHECKS OFF}
Default	{$Q-} {$OVERFLOWCHECKS OFF}
Scope	Local

Remarks

The $Q directive controls the generation of overflow checking code. In the {$Q+} state, certain integer arithmetic operations (+, -, *, Abs, Sqr, Succ, Pred, Inc, and Dec) are checked for overflow. The code for each of these integer arithmetic operations is followed by additional code that verifies that the result is within the supported range. If an overflow check fails, an EIntOverflow exception is raised (or the program is terminated if exception handling is not enabled).

The $Q switch is usually used in conjunction with the $R switch, which enables and disables the generation of range-checking code. Enabling overflow checking slows down your program and makes it somewhat larger, so use {$Q+} only for debugging.

3.1.1.44 Pentium-safe FDIV operations (Delphi)

Type	Switch
Syntax	{$U+} or {$U-} {$SAFEDIVIDE ON} or {$SAFEDIVIDE OFF}
Default	{$U-}
Scope	Local

The $U directive controls generation of floating-point code that guards against the flawed FDIV instruction exhibited by certain early Pentium processors. Windows 95, Windows NT 3.51, and later contain code which corrects the Pentium FDIV bug system-wide.

In the {$U+} state, all floating-point divisions are performed using a runtime library routine. The first time the floating-point division routine is invoked, it checks whether the processor's FDIV instruction works correctly, and updates the TestFDIV variable (declared in the System unit) accordingly. For subsequent floating-point divide operations, the value stored in TestFDIV is used to determine what action to take.

Value Meaning

-1 FDIV instruction has been tested and found to be flawed.

0 FDIV instruction has not yet been tested.

1 FDIV instruction has been tested and found to be correct.

For processors that do not exhibit the FDIV flaw, {$U+} results in only a slight performance degradation. For a flawed Pentium processor, floating-point divide operations may take up to three times longer in the {$U+} state, but they will always produce correct results.

In the {$U-} state, floating-point divide operations are performed using in-line FDIV instructions. This results in optimum speed and code size, but may produce incorrect results on flawed Pentium processors. You should use the {$U-} state only in cases where you are certain that the code is not running on a flawed Pentium processor.

3.1.1.45 NODEFINE

Type	Parameter
Syntax	{$NODEFINE identifier}

The NODEFINE directive prevents the specified symbol from being included in the header file generated for C++, while allowing some information to be output to the OBJ file. When you use NODEFINE, it is your responsibility to define any necessary types with HPPEMIT. For example:

```
type
       Temperature = type double;
       {$NODEFINE Temperature}
       {$HPPEMIT 'typedef double Temperature'}
```

3.1.1.46 NOINCLUDE (Delphi)

Type	Parameter
Syntax	{$NOINCLUDE filename}

The NOINCLUDE directive prevents the specified file from being included in header files generated for C++. For example, {$NOINCLUDE Unit1} removes #include Unit1.

3.1.1.47 Range checking

Type	Switch
Syntax	{$R+} or {$R-} {$RANGECHECKS ON} or {$RANGECHECKS OFF}
Default	{$R-} {$RANGECHECKS OFF}
Scope	Local

Remarks

The $R directive enables or disables the generation of range-checking code. In the {$R+} state, all array and string-indexing expressions are verified as being within the defined bounds, and all assignments to scalar and subrange variables are checked to be within range. If a range check fails, an ERangeError exception is raised (or the program is terminated if exception handling is not enabled).

Enabling range checking slows down your program and makes it somewhat larger.

3.1.1.48 Real48 compatibility (Delphi)

Type	Switch
Syntax	{$REALCOMPATIBILITY ON} or {$REALCOMPATIBILITY OFF}
Default	{$REALCOMPATIBILITY OFF}
Scope	Local

Remarks

In the default {**$REALCOMPATIBILITY OFF**} state, the generic Real type is equivalent to Double.

In the {**$REALCOMPATIBILITY ON**} state, Real is equivalent to Real48.

The **REALCOMPATIBILITY** switch provides backward compatibility for legacy code in which Real is used to represent the 6-byte real type now called Real48. In new code, use Real48 when you want to specify a 6-byte real. Note however, that the Real48 type is deprecated on Delphi for .NET.

Double is the preferred real type for most purposes.

3.1.1.49 Regions (Delphi and C#)

Type	Parameter
Syntax	{$REGION '<region description>'} and {$ENDREGION} [Delphi] or #region <region description> and #endregion [C#]

The **region** and **endregion** directives control the display of collapsible regions in the code editor. These directives are ignored by the compiler.

To mark code as a region, surround it with the **region** and **endregion** directives. You may include a message that will be displayed when the code is folded and hidden.

The following code shows the use of a region in Delphi:

```
{$region 'Optional text that appears when the code block is folded'}
// code that can be hidden by folding
{$endregion}
```

The following code shows the use of a region in C#:

```
#region Optional text that appears when the code block is folded
// code that can be hidden by folding
#endregion
```

3.1.1.50 Resource file (Delphi)

Type	Parameter

Syntax	{$R filename}
	{$RESOURCE filename}
	{$R *.xxx}
	{$R filename.res filename.rc}

Scope	Local

Remarks

The $R directive specifies the name of a resource file to be included in an application or library. The named file must be a Windows resource file and the default extension for filenames is .res. To specify a file name that includes a space, surround the file name with single quotation marks: {$R 'My file'}.

The ***** symbol has a special meaning in $R directives: it stands for the base name (without extension) of the source-code file where the directive occurs. Usually, an application's resource (.res) file has the same name as its project (.dpr) file; in this case, including {$R *.res} in the project file links the corresponding resource file to the application. Similarly, a form (.dfm or nfm) file usually has the same name as its unit (.pas) file; including {$R *.nfm} in the .pas file links the corresponding form file to the application.

{$R filename.res filename.rc} (where the two occurrences of 'filename' match) makes the .rc file appear in the Project Manager. When the user opens the .rc file from the Project Manager, the String Table editor is invoked.

When a {$R filename} directive is used in a unit, the specified file name is simply recorded in the resulting unit file. No checks are made at that point to ensure that the filename is correct and that it specifies an existing file.

When an application or library is linked (after compiling the program or library source file), the resource files specified in all used units as well as in the program or library itself are processed, and each resource in each resource file is copied to the executable being produced. During the resource processing phase, the linker searches for .res files in the same directory as the module containing the $R directive, and in the directories specified in the Search path input box on the Directories/Conditionals page of the Project|Options dialog box (or in the directories specified in a -R option on the `dccil` command line).

3.1.1.51 RUNONLY directive (Delphi)

Type	Switch
Syntax	{$RUNONLY ON} or {$RUNONLY OFF}
Default	{$RUNONLY OFF}
Scope	Local

Remarks

The **{$RUNONLY ON}** directive causes the package where it occurs to be compiled as runtime only. Packages compiled with **{$RUNONLY ON}** cannot be installed as design-time packages in the IDE.

Place the **$RUNONLY** directive only in package files.

See Also

Compiling packages (see page 640)

3.1.1.52 Runtime type information (Delphi)

Type	Switch
Syntax	{$M+} or {$M-} {$TYPEINFO ON} or {$TYPEINFO OFF}
Default	{$M-} {$TYPEINFO OFF}
Scope	Local

The `$M` switch directive controls generation of runtime type information (RTTI). When a class is declared in the `{$M+}` state, or is derived from a class that was declared in the `{$M+}` state, the compiler generates runtime type information for properties and events that are declared in a published section. If a class is declared in the `{$M+}` state, and is not derived from a class that was declared in the `{$M}` state, published sections are not allowed in the class. Note that if a class is forward declared, the first declaration of the class must be declared with the `$M`switch.

When the `$M` switch is used to declare an interface, the compiler generates runtime type information for all properties. That is, for interfaces, all members are treated as if they were published.

Note: The TPersistent class defined in the Classes unit of the component library is declared in the `{$M+}` state, so any class derived from TPersistent will have RTTI generated for its published sections. The component library uses the runtime type information generated for published sections to access the values of a component's properties when saving or loading form files.

Furthermore, the IDE uses a component's runtime type information to determine the list of properties to show in the Object Inspector.

Note: The IInvokable interface defined in the System unit is declared in the `{$M+}` state, so any interface derived from IInvokable will have RTTI generated. The routines in the IntfInfo unit can be used to retrieved the RTTI.

There is seldom, if ever, any need for an application to directly use the `{$M}`compiler switch.

See Also

Understanding Invokable Interfaces

$METHODINFO directive (see page 299)

3.1.1.53 Symbol declaration and cross-reference information (Delphi)

Type	Switch
Syntax	{$Y+}, {$Y-}, or {$YD}; {$REFERENCEINFO ON}, {DEFINITIONINFO OFF} or {$REFERENCEINFO OFF}, or {DEFINITIONINFO ON}
Default	{$YD} {$DEFINITIONINFO ON}
Scope	Global

Remarks

The $Y directive controls generation of symbol reference information used by the Project Manager, Code Explorer, and Code editor. This information consists of tables that provide the source-code line numbers for all declarations of and (in the {$Y+} state) references to identifiers in a module. For units, the information is recorded in the .dcu file along with the unit's object code. Symbol reference information increases the size of .dcu files, but it does not affect the size or speed of the executable program.

When a program or unit is compiled in the default {$YD} (or {DEFINITIONINFO ON}) state, the compiler records information about where each identifier is defined. For most identifiers—variables, constants, classes, and so forth—the compiler records the location of the declaration. For procedures, functions, and methods, the compiler records the location of the implementation. This enables Code editor browsing.

When a program or unit is compiled in the {$Y+} (or {REFERENCEINFO ON}) state, the compiler records information about where every identifier is used as well as where it is defined. This enables the References page of the Project Browser.

When a program or unit is compiled in the {$Y-} (or {DEFINITIONINFO OFF} or {REFERENCEINFO OFF}) state, no symbol reference information is recorded. This disables Code editor browsing and the References page of the Project Browser.

The $Y switch is usually used in conjunction with the $D and $L switches, which control generation of debug information and local symbol information. The $Y directive has no effect unless both $D and $L are enabled.

Note: Generating full cross-reference information ({$Y+}) can slow the compile/link cycle, so you should not use this except when you need the Project Manager References page.

3.1.1.54 Type-checked pointers (Delphi)

Type	Switch
Syntax	{$T+} or {$T-} {$TYPEDADDRESS ON} or {$TYPEDADDRESS OFF}
Default	{$T-} {$TYPEDADDRESS OFF}

Scope	Global

Remarks

The $T directive controls the types of pointer values generated by the @ operator and the compatibility of pointer types.

In the {$T-} state, the result of the @ operator is always an untyped pointer (Pointer) that is compatible with all other pointer types. When @ is applied to a variable reference in the {$T+} state, the result is a typed pointer that is compatible only with Pointer and with other pointers to the type of the variable.

In the {$T-} state, distinct pointer types other than Pointer are incompatible (even if they are pointers to the same type). In the {$T+} state, pointers to the same type are compatible.

3.1.1.55 UNDEF directive (Delphi)

Type	Conditional compilation
Syntax	{$UNDEF name}

Remarks

Undefines a previously defined conditional symbol. The symbol is forgotten for the remainder of the compilation of the current source file or until it reappears in a $**DEFINE** directive. The $**UNDEF** directive has no effect if name is already undefined.

Conditional symbols defined with a command-line switch or through the Project|Options dialog are reinstated at the start of compilation of each unit source file. Conditional symbols defined in a unit source file are forgotten when the compiler starts on another unit.

3.1.1.56 Unsafe Code (Delphi for .NET)

Type	Switch
Syntax	{$UNSAFECODE ON} or {$UNSAFECODE OFF}
Default	{$UNSAFECODE OFF}
Scope	Local

The **$UNSAFECODE** directive controls whether the **unsafe** keyword is accepted by the compiler. With **{$UNSAFECODE ON}**, you can mark procedures and functions with the **unsafe** keyword, for example:

```
procedure unsafeProc; unsafe;
begin
end;
```

Note: Unsafe code will not pass PEVerify

, nor will any assembly or Delphi module that calls an **unsafe** procedure or function.

3.1.1.57 Var-string checking (Delphi)

Type	Switch
Syntax	{$V+} or {$V-} {$VARSTRINGCHECKS ON} or {$VARSTRINGCHECKS OFF}

Default	{$V+} {$VARSTRINGCHECKS ON}
Scope	Local

Remarks

The **$V** directive is meaningful only for Delphi code that uses short strings, and is provided for backwards compatibility with early versions of Delphi and CodeGear Pascal.

The $V directive controls type checking on short strings passed as variable parameters. In the {$V+} state, strict type checking is performed, requiring the formal and actual parameters to be of identical string types. In the {$V-} (relaxed) state, any short string type variable is allowed as an actual parameter, even if the declared maximum length is not the same as that of the formal parameter.

3.1.1.58 Warning messages (Delphi)

Type	Switch
Syntax	{$WARN identifier ON} or {$WARN identifier OFF}
Default	{$WARN ON}
Scope	Local

Remarks

The **$WARN** directive lets you control the display of groups of warning messages. These warnings relate to symbols or units that use the hint directives, **platform**, **deprecated**, and **library**.

The identifier in the $WARN directive is optional and can have any of the following values:

SYMBOL_PLATFORM: Turns on or off all warnings about the **platform** directive on symbols in the current unit.

SYMBOL_LIBRARY: Turns on or off all warnings about the **library** directive on symbols in the current unit.

SYMBOL_DEPRECATED: Turns on or off all warnings about the **deprecated** directive on symbols in the current unit.

UNIT_DEPRECATED: Turns on or off all warnings about the **deprecated** directive applied to a unit declaration.

UNIT_LIBRARY: Turns on or off all warnings about the **library** directive in units where the **library** directive is specified.

UNIT_PLATFORM: Turns on or off all warnings about the **platform** directive in units where the **platform** directive is specified.

The only warnings that can be turned on/off using $WARN are the ones listed above.

The warnings set by the inline $WARN directive are carried for the compilation unit in which the directive appears, after which it reverts to the previous state. The warnings set by a $WARN directive take effect from that point on in the file.

The **$WARNINGS** directive also controls the generation of compiler warnings.

See Also

Declarations (↗ see page 705)

3.1.1.59 Warnings (Delphi)

Type	Switch
Syntax	{$WARNINGS ON} or {$WARNINGS OFF}
Default	{$WARNINGS ON}
Scope	Local

Remarks

The **$WARNINGS** directive controls the generation of compiler warnings. The **$WARN** directive lets control the display of groups of warning messages.

In the {**$WARNINGS ON**} state, the compiler issues warning messages when detecting uninitialized variables, missing function results, construction of abstract objects, and so on. In the {**$WARNINGS OFF**} state, the compiler generates no warning messages.

By placing code between {**$WARNINGS OFF**} and {**$WARNINGS ON**} directives, you can selectively turn off warnings that you don't care about.

Note: Note: The $WARNINGS directive only works at the procedure or function level granularity. That is, you can surround entire procedures and functions with the $WARNINGS directive, but not blocks of statements within a procedure or function.

3.1.1.60 Weak packaging

Type	Switch
Syntax	{$WEAKPACKAGEUNIT ON} or {$WEAKPACKAGEUNIT OFF}
Default	{$WEAKPACKAGEUNIT OFF}
Scope	Local

Remarks

The **$WEAKPACKAGEUNIT** directive affects the way a .dcu file is stored in a Delphi package's .dcp and .bpl files on the Win32 platform, or, analogously, how a .dcuil file is stored in the package's .dcpil and .dll files on the .NET platform. If **{$WEAKPACKAGEUNIT ON}** appears in a unit file, the compiler omits the unit from bpls or dlls when possible, and creates a non-packaged local copy of the unit when it is required by another application or package. A unit compiled with this directive is said to be "weakly packaged."

For example, suppose a package called PACK contains only one unit, UNIT1. Suppose UNIT1 does not use any further units, but it makes calls to RARE.DLL. If the **{$WEAKPACKAGEUNIT ON}** directive is inserted in UNIT1.pas before compiling, UNIT1 will not be included in PACK.BPL (or PACK.DLL on .NET); copies of RARE.DLL will not have to be distributed with PACK. However, UNIT1 will still be included in PACK.dcp (or PACK.dcpil on .NET). If UNIT1 is referenced by another package or application that uses PACK, it will be copied from PACK.dcp (or PACK.dcpil on .NET) and compiled directly into the project.

Now suppose a second unit, UNIT2, is added to PACK. Suppose that UNIT2 uses UNIT1. This time, even if PACK is compiled with **{$WEAKPACKAGEUNIT ON}** in UNIT1.pas, the compiler will include UNIT1 in PACK.BPL (or PACK.DLL on .NET). But other packages or applications that reference UNIT1 will use the (non-packaged) copy taken from PACK.dcp (or PACK.dcpil on .NET).

Note: Note: Unit files containing the **{$WEAKPACKAGEUNIT ON}** directive must not have global variables, initialization sections, or finalization sections.

The **$WEAKPACKAGEUNIT** directive is an advanced feature intended for developers who distribute their packages to other programmers. It can help to avoid distribution of infrequently used DLLs, and to eliminate conflicts among packages that may depend on the same external library.

For example, Delphi's PenWin unit references PENWIN.DLL. Most projects don't use PenWin, and most computers don't have PENWIN.DLL installed on them. For this reason, the PenWin unit is weakly packaged in VCL60 (which encapsulates many commonly-used Delphi components). When you compile a project that uses PenWin and the VCL60 package, PenWin is copied from VCL60.DCP and bound directly into your project; the resulting executable is statically linked to PENWIN.DLL.

If PenWin were not weakly packaged, two problems would arise. First, VCL60 itself would be statically linked to PENWIN.DLL, and so could not be loaded on any computer which didn't have PENWIN.DLL installed. Second, if someone tried to create a package that contained PenWin, a compiler error would result because the PenWin unit would be contained in both VCL60 and the new package. Thus, without weak packaging, PenWin could not be included in standard distributions of VCL60.

3.1.1.61 Stack frames (Delphi)

Type	Switch
Syntax	{$W+} or {$W-} {$STACKFRAMES ON} or {$STACKFRAMES OFF}
Default	{$W-} {$STACKFRAMES OFF}
Scope	Local

Remarks

The **$W** directive controls the generation of stack frames for procedures and functions. In the {**$W+**} state, stack frames are always generated for procedures and functions, even when they're not needed. In the {**$W-**} state, stack frames are only generated when they're required, as determined by the routine's use of local variables.

Some debugging tools require stack frames to be generated for all procedures and functions, but other than that you should never have a need to use the {**$W+**} state.

3.1.1.62 Writeable typed constants (Delphi)

Type	Switch
Syntax	{$J+} or {$J-} {$WRITEABLECONST ON} or {$WRITEABLECONST OFF}
Default	{$J-} {$WRITEABLECONST OFF}
Scope	Local

The **$J** directive controls whether typed constants can be modified or not. In the {**$J+**} state, typed constants can be modified, and are in essence initialized variables. In the {**$J-**} state, typed constants are truly constant, and any attempt to modify a typed constant causes the compiler to report an error.

Writeable consts refers to the use of a typed const as a variable modifiable at runtime. For example:

```
const
       foo: Integer = 12;
    begin
        foo := 14;
    end.
```

With $WRITEABLECONST OFF, this code produces a compile error on the assignment to the foo variable in the begin..end block. To fix it, change the const declaration to a var declaration.

In early versions of Delphi and CodeGear Pascal, typed constants were always writeable, corresponding to the {**$J+**} state. Old source code that uses writeable typed constants must be compiled in the {**$J+**} state, but for new applications it is recommended that you use initialized variables and compile your code in the {**$J-**} state.

3.1.1.63 PE (portable executable) header flags (Delphi)

Type	Flag
Syntax	{$SetPEFlags <integer expression>} {$SetPEOptFlags <integer expression>}
Scope	Local

Microsoft relies on PE (portable executable) header flags to allow an application to indicate compatiblity with OS services or request advanced OS services. These directives provide powerful options for tuning your applications on high-end NT systems.

Warning: There is no error checking or masking of bit values specified by these directives. If you set the wrong combination of bits, you could corrupt your executable file.

These directives allow you to set flag bits in the PE file header Characteristics field and PE file optional header DLLCharacteristics field, respectively. Most of the Characteristics flags, set using $SetPEFlags, are specifically for object files and libraries. DLLCharacteristics, set using $SetPEOptFlags, are flags that describe when to call a DLL's entry point.

The <integer expression> in these directives can include Delphi constant identifiers, such as the IMAGE_FILE_xxxx constants defined in Windows.pas. Multiple constants should be OR'd together.

You can include these directives in source code multiple times. The flag values specified by multiple directives are strictly cumulative: if the first occurrence of the directive sets $03 and the second occurrence sets $10, the value written to the executable file at link time will be $13 (plus whatever bits the linker normally sets in the PE flag fields).

These directives only affect the output file if included in source code prior to linking. This means you should place these directives in a .dpr or .dpk file, not in a regular unit. Like the exe description directive, it's not an error to place these directives in unit source code, but these directives in unit source will not affect the output file (exe or dll) unless the unit source is recompiled at the time the output file is linked.

3.1.1.64 Reserved address space for resources (Delphi)

Type	OS	Parameter
Syntax	Linux	{$M reservedbytes} {$RESOURCERESERVE reservedbytes}
Default		{$M 1048576}
Scope		Global

Remarks

This directive is used in Linux programming only. For information about the $M ($MINSTACKSIZE and $MAXSTACKSIZE) directives in Windows, see the topic on Memory Allocation Sizes.

Use the **$M** directive to increase or decrease the amount of extra address space reserved for resources.

By default, the compiler reserves 1MB of address space, in addition to what the application actually uses at link time, for resources. This extra address space is provided to accommodate localized versions of the application that incorporate larger resource files than the original version. As long as there is sufficient reserved address space, you won't have to relink the entire executable to produce a localized version.

For maximum portability between Windows and Linux platforms, you should use the long form of this directive $**RESOURCERESERVE** and not $**M**.

See Also

Memory allocation sizes

3.1.2 Delphi Compiler Errors

The following topics describe the various types of compiler errors and warnings, along with resolutions to many issues you may face while using this product.

Topics

Name	Description
Error Messages (↗ see page 311)	This section lists all the Delphi compiler error and warning messages of RAD Studio.
Delphi Runtime Errors (↗ see page 509)	Certain errors at runtime cause Delphi programs to display an error message and terminate.
I/O Errors (↗ see page 510)	I/O errors cause an exception to be thrown if a statement is compiled in the {$I+} state. (If the application does not include the SysUtils class, the exception causes the application to terminate).
Fatal errors (↗ see page 511)	These errors always immediately terminate the program.
Operating system errors (↗ see page 512)	All errors other than I/O errors and fatal errors are reported with the error codes returned by the operating system.

3.1.2.1 Error Messages

This section lists all the Delphi compiler error and warning messages of RAD Studio.

Topics

Name	Description
DisposeCount cannot be declared in classes with destructors (↗ see page 336)	No further Help is available for this message or warning.
E2190: Thread local variables cannot be ABSOLUTE (↗ see page 336)	A thread local variable cannot refer to another variable, nor can it reference an absolute memory address.
E2249: Cannot use absolute variables when compiling to byte code (↗ see page 336)	The use of absolute variables is prohibited when compiling to byte code.
E2373: Call to abstract method %s.%s (↗ see page 336)	No further information is available for this error or warning.
E2371: ABSTRACT and FINAL cannot be used together (↗ see page 337)	A class cannot be both final and abstract. Final is a restrictive modifier used to prevent extension of a class (or prevent overrides on methods), while the abstract modifier signals the intention to use a class as a base class.
E2136: No definition for abstract method '%s' allowed (↗ see page 337)	You have declared <name> to be abstract, but the compiler has found a definition for the method in the source file. It is illegal to provide a definition for an abstract declaration.
E2167: Abstract methods must be virtual or dynamic (↗ see page 337)	When declaring an abstract method in a base class, it must either be of regular virtual or dynamic virtual type.
E2383: ABSTRACT and SEALED cannot be used together (↗ see page 338)	A class cannot be both sealed and abstract. The sealed modifier is used to prevent inheritance of a class, while the abstract modifier signals the intention to use a class as a base class.
E2247: Cannot take the address when compiling to byte code (↗ see page 338)	The address-of operator, @, cannot be used when compiling to byte code.
E2251: Ambiguous overloaded call to '%s' (↗ see page 338)	Based on the current overload list for the specified function, and the programmed invocation, the compiler is unable to determine which version of the procedure should be invoked.
E2099: Overflow in conversion or arithmetic operation (↗ see page 339)	The compiler has detected an overflow in an arithmetic expression: the result of the expression is too large to be represented in 32 bits. Check your computations to ensure that the value can be represented by the computer hardware.

E2307: NEW standard function expects a dynamic array type identifier (↗ see page 339)	No further information is available for this error or warning.
E2308: Need to specify at least one dimension for NEW of dynamic array (↗ see page 339)	No further information is available for this error or warning.
E2246: Need to specify at least one dimension for SetLength of dynamic array (↗ see page 339)	The standard procedure SetLength has been called to alter the length of a dynamic array, but no array dimensions have been specified.
E2081: Assignment to FOR-Loop variable '%s' (↗ see page 340)	It is illegal to assign a value to the for loop control variable inside the for loop. If the purpose is to leave the loop prematurely, use a break or goto statement.
W1017: Assignment to typed constant '%s' (↗ see page 340)	This warning message is currently unused.
E2117: 486/487 instructions not enabled (↗ see page 340)	You should not receive this error as 486 instructions are always enabled.
E2116: Invalid combination of opcode and operands (↗ see page 340)	You have specified an inline assembler statement which is not correct.
E2109: Constant expected (↗ see page 341)	The inline assembler was expecting to find a constant but did not find one.
E2118: Division by zero (↗ see page 341)	The inline assembler has encountered an expression which results in a division by zero.
E2119: Structure field identifier expected (↗ see page 341)	The inline assembler recognized an identifier on the right side of a '.', but it was not a field of the record found on the left side of the '.'. One common, yet difficult to realize, error of this sort is to use a record with a field called 'ch' - the inline assembler will always interpret 'ch' to be a register name.
E2108: Memory reference expected (↗ see page 342)	The inline assembler has expected to find a memory reference expression but did not find one. Ensure that the offending statement is indeed a memory reference.
E2115: Error in numeric constant (↗ see page 342)	The inline assembler has found an error in the numeric constant you entered.
E2107: Operand size mismatch (↗ see page 343)	The size required by the instruction operand does not match the size given.
E2113: Numeric overflow (↗ see page 343)	The inline assembler has detected a numeric overflow in one of your expressions.
E2112: Invalid register combination (↗ see page 344)	You have specified an illegal combination of registers in a inline assembler statement. Please refer to an assembly language guide for more information on addressing modes allowed on the Intel 80x86 family.
E2111: Cannot add or subtract relocatable symbols (↗ see page 344)	The inline assembler is not able to add or subtract memory address which may be changed by the linker.
E2106: Inline assembler stack overflow (↗ see page 345)	Your inline assembler code has exceeded the capacity of the inline assembler. Contact CodeGear if you encounter this error.
E2114: String constant too long (↗ see page 345)	The inline assembler has not found the end of the string that you specified. The most likely cause is a misplaced closing quote.
E2105: Inline assembler syntax error (↗ see page 345)	You have entered an expression which the inline assembler is unable to interpret as a valid assembly instruction.
E2110: Type expected (↗ see page 346)	Contact CodeGear if you receive this error.
E2448: An attribute argument must be a constant expression, typeof expression or array constructor (↗ see page 346)	The Common Language Runtime specifies that an attribute argument must be a constant expression, a typeof expression or an array creation expression. Attribute arguments cannot be global variables, for example. Attribute instances are constructed at compile-time and incorporated into the assembly metadata, so no run-time information can be used to construct them.
E2045: Bad object file format: '%s' (↗ see page 346)	This error occurs if an object file loaded with a $L or $LINK directive is not of the correct format. Several restrictions must be met: • Check the naming restrictions on segment names in the help file • Not more than 10 segments • Not more than 255 external symbols • Not more than 50 local names in LNAMES records • LEDATA and LIDATA records must be in offset order • No THREAD subrecords are supported in FIXU32 records • Only 32-bit offsets can be fixed up • Only segment and self relative fixups • Target of a fixup must be a segment, a group or an EXTDEF • Object... more (↗ see page 346)

x1028: Bad global symbol definition: '%s' in object file '%s' (see page 346)	This warning is given when an object file linked in with a $L or $LINK directive contains a definition for a symbol that was not declared in Delphi as an external procedure, but as something else (e.g. a variable). The definition in the object will be ignored in this case.
E2160: Type not allowed in OLE Automation call (see page 346)	If a data type cannot be converted by the compiler into a Variant, then it is not allowed in an OLE automation call.
E2188: Published property '%s' cannot be of type %s (see page 347)	Published properties must be an ordinal type, Single, Double, Extended, Comp, a string type, a set type which fits in 32 bits, or a method pointer type. When any other property type is encountered in a published section, the compiler will remove the published attribute -$M+
E2055: Illegal type in Read/Readln statement (see page 348)	This error occurs when you try to read a variable in a Read or Readln that is not of a legal type. Check the type of the variable and make sure you are not missing a dereferencing, indexing or field selection operator.
E2053: Syntax error in real number (see page 348)	This error message occurs if the compiler finds the beginning of a scale factor (an 'E' or 'e' character) in a number, but no digits follow it.
E2104: Bad relocation encountered in object file '%s' (see page 348)	You are trying to link object modules into your program with the $L compiler directive. However, the object file is too complex for the compiler to handle. For example, you may be trying to link in a C++ object file. This is not supported.
E2158: %s unit out of date or corrupted: missing '%s' (see page 349)	The compiler is looking for a special function which resides in System.dcu but could not find it. Your System unit is either corrupted or obsolete. Make sure there are no conflicts in your library search path which can point to another System.dcu. Try reinstalling System.dcu. If neither of these solutions work, contact CodeGear Developer Support.
E2159: %s unit out of date or corrupted: missing '%s.%s' (see page 349)	The compiler failed to find a special function in System, indicating that the unit found in your search paths is either corrupted or obsolete.
E2150: Bad argument type in variable type array constructor (see page 349)	You are attempting to construct an array using a type which is not allowed in variable arrays.
E2281: Type not allowed in Variant Dispatch call (see page 350)	This message indicates that you are trying to make a method call and are passing a type that the compiler does not know how to marshall. Variants can hold interfaces, but the interfaces can marshall only certain types. On Windows, Delphi supports COM and SOAP interfaces and can call types that these interfaces can marshall.
E2054: Illegal type in Write/Writeln statement (see page 350)	This error occurs when you try to output a type in a Write or Writeln statement that is not legal.
E2297: Procedure definition must be ILCODE calling convention (see page 350)	.NET managed code can only use the ILCODE calling convention.
E2050: Statements not allowed in interface part (see page 350)	The interface part of a unit can only contain declarations, not statements. Move the bodies of procedures to the implementation part.
x1012: Constant expression violates subrange bounds (see page 351)	This error message occurs when the compiler can determine that a constant is outside the legal range. This can occur for instance if you assign a constant to a variable of subrange type.
E2097: BREAK or CONTINUE outside of loop (see page 351)	The compiler has found a BREAK or CONTINUE statement which is not contained inside a WHILE or REPEAT loop. These two constructs are only legal in loops.
E2309: Attribute - Known attribute named argument cannot be an array (see page 352)	No further information is available for this error or warning.
E2310: Attribute - A custom marshaler requires the custom marshaler type (see page 352)	No further information is available for this error or warning.
E2327: Linker error while emitting attribute '%s' for '%s' (see page 352)	No further information is available for this error or warning.
E2311: Attribute - MarshalAs fixed string requires a size (see page 352)	No further information is available for this error or warning.
E2312: Attribute - Invalid argument to a known attribute (see page 352)	No further information is available for this error or warning.
E2313: Attribute - Known attribute cannot specify properties (see page 352)	No further information is available for this error or warning.
E2314: Attribute - The MarshalAs attribute has fields set that are not valid for the specified unmanaged type (see page 352)	No further information is available for this error or warning.
E2315: Attribute - Known custom attribute on invalid target (see page 352)	No further information is available for this error or warning.
E2316: Attribute - The format of the GUID was invalid (see page 353)	No further information is available for this error or warning.
E2317: Attribute - Known custom attribute had invalid value (see page 353)	No further information is available for this error or warning.
E2318: Attribute - The MarshalAs constant size cannot be negative (see page 353)	No further information is available for this error or warning.
E2319: Attribute - The MarshalAs parameter index cannot be negative (see page 353)	No further information is available for this error or warning.
E2320: Attribute - The specified unmanaged type is only valid on fields (see page 353)	No further information is available for this error or warning.

E2321: Attribute - Known custom attribute has repeated named argument (↗ see page 353)	No further information is available for this error or warning.
E2322: Attribute - Unexpected type in known attribute (↗ see page 353)	No further information is available for this error or warning.
E2323: Attribute - Unrecognized argument to a known custom attribute (↗ see page 353)	No further information is available for this error or warning.
E2324: Attribute - Known attribute named argument doesn't support variant (↗ see page 353)	No further information is available for this error or warning.
E2222: $WEAKPACKAGEUNIT & $DENYPACKAGEUNIT both specified (↗ see page 354)	It is not legal to specify both $WEAKPACKAGEUNIT and $DENYPACKAGEUNIT. Correct the source code and recompile.
E2276: Identifier '%s' cannot be exported (↗ see page 354)	This message indicates that you are trying to export a function or procedure that is tagged with the local directive. You also, cannot export threadvars and you would receive this message if you try to do so.
E2071: This type cannot be initialized (↗ see page 354)	File types (including type Text), and the type Variant cannot be initialized, that is, you cannot declare typed constants or initialized variables of these types.
E2374: Cannot make unique type from %s (↗ see page 354)	No further information is available for this error or warning.
E2223: $DENYPACKAGEUNIT '%s' cannot be put into a package (↗ see page 354)	You are attempting to put a unit which was compiled with $DENYPACKAGEUNIT into a package. It is not possible to put a unit compiled with the $DENYPACKAGEUNIT direction into a package.
E2217: Published field '%s' not a class or interface type (↗ see page 354)	An attempt has been made to publish a field in a class which is not a class nor interface type.
E2218: Published method '%s' contains an unpublishable type (↗ see page 355)	This message is not used in dccil. The message applies only to Win32 compilations, where it indicates that a parameter or function result type in the method is not a publishable type.
E2278: Cannot take address of local symbol %s (↗ see page 355)	This message occurs when you try to call a symbol from within a procedure or function that has been tagged with the local directive. The local directive, which marks routines as unavailable for export, is platform-specific and has no effect in Windows programming. On Linux, the local directive is used for routines that are compiled into a library but are not exported. This directive can be specified for standalone procedures and functions, but not for methods. A routine declared with local, for example,
E2392: Can't generate required accessor method(s) for property %s.%s due to name conflict with existing symbol %s in the same scope (↗ see page 355)	The CLR requires that property accessors be methods, not fields. The Delphi language allows you to specify fields as property accessors. The Delphi compiler will generate the necessary methods behind the scenes. CLS recommends a specific naming convention for property accessor methods: get_propname and set_propname. If the accessors for a property are not methods, or if the given methods do not match the CLS name pattern, the Delphi compiler will attempt to generate methods with CLS conforming names. If a method already exists in the class that matches the CLS name pattern, but it is not associated with the particular... more (↗ see page 355)
E2126: Cannot BREAK, CONTINUE or EXIT out of a FINALLY clause (↗ see page 356)	Because a FINALLY clause may be entered and exited through the exception handling mechanism or through normal program control, the explicit control flow of your program may not be followed. When the FINALLY is entered through the exception handling mechanism, it is not possible to exit the clause with BREAK, CONTINUE, or EXIT - when the finally clause is being executed by the exception handling system, control must return to the exception handling system.
W1018: Case label outside of range of case expression (↗ see page 356)	You have provided a label inside a case statement which cannot be produced by the case statement control variable. -W
E2326: Attribute '%s' can only be used once per target (↗ see page 357)	This attribute can only be used once per target Attributes and their descendants may be declared with an AttributeUsage Attribute which describes how a custom Attribute may be used. If the use of an attribute violates AttributeUsage.allowmultiple then this error will be raised.
E2325: Attribute '%s' is not valid on this target (↗ see page 357)	Attribute is not valid on this target. Attributes and their descendants may be declared with an AttributeUsage Attribute which describes how a custom Attribute may be used. If the use of an attribute violates AttributeUsage.validon property then this error will be raised. AttributeUsage.validon specifies the application element that this attribute may be applied to.
E2358: Class constructors not allowed in class helpers (↗ see page 357)	No further information is available for this error or warning.
E2360: Class constructors cannot have parameters (↗ see page 358)	No further information is available for this error or warning.
E2340: Metadata - Data too large (↗ see page 358)	No further information is available for this error or warning.
E2343: Metadata - Primary key column may not allow the null value (↗ see page 358)	No further information is available for this error or warning.
E2341: Metadata - Column cannot be changed (↗ see page 358)	No further information is available for this error or warning.
E2342: Metadata - Too many RID or primary key columns, 1 is max (↗ see page 358)	No further information is available for this error or warning.
E2329: Metadata - Error occured during a read (↗ see page 358)	No further information is available for this error or warning.
E2330: Metadata - Error occured during a write (↗ see page 358)	No further information is available for this error or warning.

E2334: Metadata - Old version error (see page 358)	No further information is available for this error or warning.
E2331: Metadata - File is read only (see page 358)	No further information is available for this error or warning.
E2339: Metadata - The importing scope is not compatible with the emitting scope (see page 358)	No further information is available for this error or warning.
E2332: Metadata - An ill-formed name was given (see page 358)	No further information is available for this error or warning.
E2337: Metadata - There isn't .CLB data in the memory or stream (see page 359)	No further information is available for this error or warning.
E2338: Metadata - Database is read only (see page 359)	No further information is available for this error or warning.
E2335: Metadata - A shared mem open failed to open at the originally (see page 359)	No further information is available for this error or warning.
E2336: Metadata - Create of shared memory failed. A memory mapping of the same name already exists (see page 359)	No further information is available for this error or warning.
E2344: Metadata - Data too large (see page 359)	No further information is available for this error or warning.
E2333: Metadata - Data value was truncated (see page 359)	No further information is available for this error or warning.
F2047: Circular unit reference to '%s' (see page 359)	One or more units use each other in their interface parts. As the compiler has to translate the interface part of a unit before any other unit can use it, the compiler must be able to find a compilation order for the interface parts of the units. Check whether all the units in the uses clauses are really necessary, and whether some can be moved to the implementation part of a unit instead.
E2123: PROCEDURE, FUNCTION, PROPERTY, or VAR expected (see page 360)	The tokens that follow "class" in a member declaration inside a class type are limited to procedure, function, var, and property.
E2061: Local class or interface types not allowed (see page 360)	Corresponds to object_local in previous compilers. Class and interface types cannot be declared inside a procedure body.
E2435: Class member declarations not allowed in anonymous record or local record type (see page 360)	Record types that are declared in local scopes or declared in-place in variable declarations can only contain field declarations. For advanced features in record types (such as methods, properties, and nested types), the record type must be an explicitly declared global type.
E2060: Class and interface types only allowed in type section (see page 360)	Class or interface types must always be declared with an explicit type declaration in a type section. Unlike record types, they cannot be anonymous. The main reason for this is that there would be no way you could declare the methods of that type (since there is no type name). Incorrect (attempting to declare a class type within a variable declaration):
E2355: Class property accessor must be a class field or class static method (see page 361)	No further information is available for this error or warning.
E2128: %s clause expected, but %s found (see page 361)	The compiler was, due to the Delphi language syntax, expecting to find a clause1 in your program, but instead found clause2.
E2401: Failure loading .NET Framework %s: %08X (see page 361)	No further information is available for this error or warning.
x2421: Imported identifier '%s' conflicts with '%s' in '%s' (see page 361)	When importing type information from a .NET assembly, the compiler may encounter symbols that do not conform to CLS specifications. One example of this is case-sensitive versus case-insensitive identifiers. Another example is having a property in a class with the same name as a method or field in the same class. This error message indicates that same-named symbols were found in the same scope (members of the same class or interface) in an imported assembly and that only one of them will be accessible from Delphi syntax.
E2422: Imported identifier '%s' conflicts with '%s' in namespace '%s' (see page 361)	When importing type information from a .NET assembly, the compiler may encounter symbols that do not conform to CLS specifications. One example of this is case-sensitive versus case-insensitive identifiers. Another example is having a property in a class with the same name as a method or field in the same class. This error message indicates that same-named symbols were found in the same scope (members of the same class or interface) in an imported assembly and that only one of them will be accessible from Delphi syntax.
H2384: CLS: overriding virtual method '%s.%s' visibility (%s) must match base class '%s' (%s) (see page 362)	No further information is available for this error or warning.
E2431: for-in statement cannot operate on collection type '%s' because '%s' does not contain a member for '%s', or it is inaccessible (see page 362)	A for-in statement can only operate on the following collection types: • Primitive types that the compiler recognizes, such as arrays, sets or strings • Types that implement IEnumerable • Types that implement the GetEnumerator pattern as documented in the Delphi Language Guide Ensure that the specified type meets these requirements.

W1024: Combining signed and unsigned types - widened both operands (↗ see page 362)	To mathematically combine signed and unsigned types correctly the compiler must promote both operands to the next larger size data type and then perform the combination. To see why this is necessary, consider two operands, an Integer with the value -128 and a Cardinal with the value 130. The Cardinal type has one more digit of precision than the Integer type, and thus comparing the two values cannot accurately be performed in only 32 bits. The proper solution for the compiler is to promote both these types to a larger, common, size and then to perform the comparison. The compiler... more (↗ see page 362)
E2008: Incompatible types (↗ see page 363)	This error message occurs when the compiler expected two types to be compatible (meaning very similar), but in fact, they turned out to be different. This error occurs in many different situations - for example when a read or write clause in a property mentions a method whose parameter list does not match the property, or when a parameter to a standard procedure or function is of the wrong type. This error can also occur when two units both declare a type of the same name. When a procedure from an imported unit has a parameter of the same-named type,... more (↗ see page 363)
E2009: Incompatible types: '%s' (↗ see page 364)	The compiler has detected a difference between the declaration and use of a procedure.
E2010: Incompatible types: '%s' and '%s' (↗ see page 365)	This error message results when the compiler expected two types to be compatible (or similar), but they turned out to be different.
W1023: Comparing signed and unsigned types - widened both operands (↗ see page 365)	To compare signed and unsigned types correctly the compiler must promote both operands to the next larger size data type. To see why this is necessary, consider two operands, a Shortint with the value -128 and a Byte with the value 130. The Byte type has one more digit of precision than the Shortint type, and thus comparing the two values cannot accurately be performed in only 8 bits. The proper solution for the compiler is to promote both these types to a larger, common, size and then to perform the comparison.
W1021: Comparison always evaluates to False (↗ see page 366)	The compiler has determined that the expression will always evaluate to False. This most often can be the result of a boundary test against a specific variable type, for example, a Integer against $80000000. In versions of the Delphi compiler prior to 12.0, the hexadecimal constant $80000000 would have been a negative Integer value, but with the introduction of the int64 type, this same constant now becomes a positive int64 type. As a result, comparisons of this constant against Integer variables will no longer behave as they once did. As this is a warning rather than an error, there is... more (↗ see page 366)
W1022: Comparison always evaluates to True (↗ see page 366)	The compiler has determined that the expression will always evaluate to true. This most often can be the result of a boundary test against a specific variable type, for example, a Integer against $80000000. In versions of the CodeGear Pascal compiler prior to 12.0, the hexadecimal constant $80000000 would have been a negative Integer value, but with the introduction of the int64 type, this same constant now becomes a positive int64 type. As a result, comparisons of this constant against Integer variables will no longer behave as they once did. As this is a warning rather than an error, there... more (↗ see page 366)
E2026: Constant expression expected (↗ see page 367)	The compiler expected a constant expression here, but the expression it found turned out not to be constant.
E2192: Constants cannot be used as open array arguments (↗ see page 367)	Open array arguments must be supplied with an actual array variable, a constructed array or a single variable of the argument's element type.
E2007: Constant or type identifier expected (↗ see page 368)	This error message occurs when the compiler expects a type, but finds a symbol that is neither a constant (a constant could start a subrange type), nor a type identifier.
E2197: Constant object cannot be passed as var parameter (↗ see page 368)	This error message is reserved.
E2177: Constructors and destructors not allowed in OLE automation section (↗ see page 368)	You have incorrectly tried to put a constructor or destructor into the 'automated' section of a class declaration.
x1020: Constructing instance of '%s' containing abstract method '%s.%s' (↗ see page 369)	The code you are compiling is constructing instances of classes which contain abstract methods.
E2402: Constructing instance of abstract class '%s' (↗ see page 370)	No further information is available for this error or warning.
E2437: Constant declarations not allowed in anonymous record or local record type (↗ see page 370)	Record types that are declared in local scopes or declared in-place in variable declarations can only contain field declarations. For advanced features in record types (such as methods, properties, and nested types), the record type must be an explicitly declared global type.
E2241: C++ obj files must be generated (-jp) (↗ see page 370)	Because of the language features used, standard C object files cannot be generated for this unit. You must generate C++ object files.
E2412: CREATE expected (↗ see page 370)	No further information is available for this error or warning.

E2306: 'Self' is initialized more than once (see page 370)	An inherited constructor has been initialized multiple times.
E2304: 'Self' is uninitialized. An inherited constructor must be called (see page 370)	In Delphi for .NET, a constructor must always call an inherited constructor before it may access or initialize any inherited class members. The compiler generates an error if your constructor code does not call the inherited constructor (a valid situation in Delphi for Win32), but it is important to examine your constructors to make sure that you do not access any inherited class fields, directly or indirectly, before the call to the inherited constructor. **Note:** A constructor can initialize fields from its own class, prior to calling the inherited constructor. **Example:** The class,
E2305: 'Self' might not have been initialized (see page 371)	In Delphi for .NET, a constructor must always call an inherited constructor before it may access or initialize any inherited class members. The compiler generates an error if your constructor code does not call the inherited constructor (a valid situation in Delphi for Win32), but it is important to examine your constructors to make sure that you do not access any inherited class fields, directly or indirectly, before the call to the inherited constructor. **Note:** A constructor can initialize fields from its own class, prior to calling the inherited constructor.
E2302: 'Self' is uninitialized. An inherited constructor must be called before accessing ancestor field '%s' (see page 371)	In Delphi for .NET, a constructor must always call an inherited constructor before it may access or initialize any inherited class members. The compiler generates an error if your constructor code does not call the inherited constructor (a valid situation in Delphi for Win32), but it is important to examine your constructors to make sure that you do not access any inherited class fields, directly or indirectly, before the call to the inherited constructor. **Note:** A constructor can initialize fields from its own class, prior to calling the inherited constructor.
E2303: 'Self' is uninitialized. An inherited constructor must be called before calling ancestor method '%s' (see page 371)	In Delphi for .NET, a constructor must always call an inherited constructor before it may access or initialize any inherited class members. The compiler generates an error if your constructor code does not call the inherited constructor (a valid situation in Delphi for Win32), but it is important to examine your constructors to make sure that you do not access any inherited class fields, directly or indirectly, before the call to the inherited constructor. **Note:** A constructor can initialize fields from its own class, prior to calling the inherited constructor.
E2286: Coverage library name is too long: %s (see page 371)	This message is not used in this product.
H2455: Narrowing given wide string constant lost information (see page 371)	Any character in a WideString constant with ordinal value greater than 127 may be replaced with "?" if the WideChar is not representable in the current locale codepage.
H2451: Narrowing given WideChar constant (#$%04X) to AnsiChar lost information (see page 371)	An AnsiChar can only represent the first 256 values in a WideChar, so the second byte of the WideChar is lost when converting it to an AnsiChar. You may wish to use WideChar instead of AnsiChar to avoid information loss.
E2238: Default value required for '%s' (see page 371)	When using default parameters a list of parameters followed by a type is not allowed; you must specify each variable and its default value individually.
E2237: Parameter '%s' not allowed here due to default value (see page 372)	When using default parameters a list of parameters followed by a type is not allowed; you must specify each variable and its default value individually.
E2132: Default property must be an array property (see page 373)	The default property which you have specified for the class is not an array property. Default properties are required to be array properties.
E2268: Parameters of this type cannot have default values (see page 373)	The default parameter mechanism incorporated into the Delphi compiler allows only simple types to be initialized in this manner. You have attempted to use a type that is not supported.
E2239: Default parameter '%s' must be by-value or const (see page 374)	Parameters which are given default values cannot be passed by reference.
E2131: Class already has a default property (see page 374)	You have tried to assign a default property to a class which already has defined a default property.
E2146: Default values must be of ordinal, pointer or small set type (see page 375)	You have declared a property containing a default clause, but the type property type is incompatible with default values.
F2087: System unit incompatible with trial version (see page 376)	You are using a trial version of the software. It is incompatible with the application you are trying to run.
E2144: Destination is inaccessible (see page 376)	The address to which you are attempting to put a value is inaccessible from within the IDE.
E2453: Destination cannot be assigned to (see page 376)	The integrated debugger has determined that your assignment is not valid in the current context.
E2290: Cannot mix destructors with IDisposable (see page 376)	The compiler will generate IDisposable support for a class that declares a destructor override named "Destroy". You cannot manually implement IDisposable and implement a destructor on the same class.

F2446: Unit '%s' is compiled with unit '%s' in '%s' but different version '%s' found (see page 376)	This error occurs if a unit must be recompiled to take in changes to another unit, but the source for the unit that needs recompilation is not found. **Note:** This error message may be experienced when using inline functions. Expansion of an inline function exposes its implementation to all units that call the function. When a function is inline, modifications to that function must be reflected with a recompile of every unit that uses that function. This is true even if all of the modifications occur in the implementation section. This is one way in which inlining can make your... more (see page 376)
E2210: '%s' directive not allowed in in interface type (see page 376)	A directive was encountered during the parsing of an interface which is not allowed.
E2228: A dispinterface type cannot have an ancestor interface (see page 377)	An interface type specified with dispinterface cannot specify an ancestor interface.
E2230: Methods of dispinterface types cannot specify directives (see page 377)	Methods declared in a dispinterface type cannot specify any calling convention directives.
E2229: A dispinterface type requires an interface identification (see page 378)	When using dispinterface types, you must always be sure to include a GUID specification for them.
E2183: Dispid clause only allowed in OLE automation section (see page 378)	A dispid has been given to a property which is not in an automated section.
E2274: property attribute 'label' cannot be used in dispinterface (see page 379)	You have added a label to a property defined in a dispinterface, but this is disallowed by the language definition.
E2080: Procedure DISPOSE needs destructor (see page 380)	This error message is issued when an identifier given in the parameter list to Dispose is not a destructor.
E2414: Disposed_ cannot be declared in classes with destructors (see page 381)	Disposed_ cannot be declared in classes with destructors. If a class implements the IDispose interface the compiler generates a field called Disposed_ to determine whether or not the IDispose.Dispose method has already been called.
E2098: Division by zero (see page 381)	The compiler has detected a constant division by zero in your program. Check your constant expressions and respecify them so that a division by zero error will not occur.
E2293: Cannot have both a DLLImport attribute and an external or calling convention directive (see page 381)	The compiler emits DLLImport attributes internally for external function declarations. This error is raised if you declare your own DLLImport attribute on a function and use the external name clause on the function.
E2027: Duplicate tag value (see page 381)	This error message is given when a constant appears more than once in the declaration of a variant record.
E2399: Namespace conflicts with unit name '%s' (see page 381)	No further information is available for this error or warning.
E2030: Duplicate case label (see page 381)	This error message occurs when there is more than one case label with a given value in a case statement.
W1029: Duplicate %s '%s' with identical parameters will be inacessible from C++ (see page 382)	An object file is being generated and Two, differently named, constructors or destructors with identical parameter lists have been created; they will be inaccessible if the code is translated to an HPP file because constructor and destructor names are converted to the class name. In C++ these duplicate declarations will appear to be the same function.
E2180: Dispid '%d' already used by '%s' (see page 383)	An attempt to use a dispid which is already assigned to another member of this class.
E2301: Method '%s' with identical parameters and result type already exists (see page 384)	Within a class, you cannot publish multiple overloaded methods with the same name. Maintenance of runtime type information requires a unique name for each published member.
E2257: Duplicate implements clause for interface '%s' (see page 384)	The compiler has encountered two different property declarations which claim to implement the same interface. An interface may be implemented by only one property.
E2447: Duplicate symbol '%s' defined in namespace '%s' by '%s' and '%s' (see page 385)	This error occurs when symbols from separate units are combined into a common namespace, and the same symbol name is in both units. In previous versions of Delphi, these units may have compiled without error, because symbol scope was defined by the unit alone. In RAD Studio, units must be inserted into namespaces when generating the IL metadata. This may cause separate units to be be combined into a single namespace. To resolve this problem, you may wish to rename one of the symbols in the two units, alias one of the symbols to the other, or change the unit... more (see page 385)
E2140: Duplicate message method index (see page 385)	You have specified an index for a dynamic method which is already used by another dynamic method.
E2252: Method '%s' with identical parameters already exists (see page 386)	A method with an identical signature already exists in the data type.
E2266: Only one of a set of overloaded methods can be published (see page 386)	Only one member of a set of overloaded functions may be published because the RTTI generated for procedures only contains the name.
E2285: Duplicate resource id: type %d id %d (see page 388)	A resource linked into the project has the same type and name, or same type and resource ID, as another resource linked into the project. (In Delphi, duplicate resources are ignored with a warning. In Kylix, duplicates cause an error.)

E2407: Duplicate resource identifier %s found in unit %s(%s) and %s(%s) (↗ see page 388)	No further information is available for this error or warning.
E2284: Duplicate resource name: type %d '%s' (↗ see page 388)	A resource linked into the project has the same type and name, or same type and resource ID, as another resource linked into the project. (In Delphi, duplicate resources are ignored with a warning. In Kylix, duplicates cause an error.)
E2429: Duplicate implementation for 'set of %s' in this scope (↗ see page 388)	To avoid this error, declare an explicit set type identifier instead of using in-place anonymous set expressions.
W1051: Duplicate symbol names in namespace. Using '%s.%s' found in %s. Ignoring duplicate in %s (↗ see page 388)	No further information is available for this error or warning.
E2413: Dynamic array type needed (↗ see page 388)	No further information is available for this error or warning.
E2178: Dynamic methods and message handlers not allowed in OLE automation section (↗ see page 388)	You have incorrectly put a dynamic or message method into an 'automated' section of a class declaration.
E2378: Error while converting resource %s (↗ see page 389)	No further information is available for this error or warning.
E2385: Error while signing assembly (↗ see page 389)	No further information is available for this error or warning.
E2125: EXCEPT or FINALLY expected (↗ see page 389)	The compiler was expecting to find a FINALLY or EXCEPT keyword, during the processing of exception handling code, but did not find either.
E2029: %s expected but %s found (↗ see page 390)	This error message appears for syntax errors. There is probably a typo in the source, or something was left out. When the error occurs at the beginning of a line, the actual error is often on the previous line.
E2191: EXPORTS allowed only at global scope (↗ see page 390)	An EXPORTS clause has been encountered in the program source at a non-global scope.
E2143: Expression has no value (↗ see page 390)	You have attempted to assign the result of an expression, which did not produce a value, to a variable.
E2353: Cannot extend sealed class '%s' (↗ see page 391)	The sealed modifier is used to prevent inheritance (and thus extension) of a class.
E2078: Procedure FAIL only allowed in constructor (↗ see page 391)	The standard procedure Fail can only be called from within a constructor - it is illegal otherwise.
E2169: Field definition not allowed after methods or properties (↗ see page 391)	You have attempted to add more fields to a class after the first method or property declaration has been encountered. You must place all field definitions before methods and properties.
E2175: Field definition not allowed in OLE automation section (↗ see page 391)	You have tried to place a field definition in an OLE automation section of a class declaration. Only properties and methods may be declared in an 'automated' section.
E2124: Instance member '%s' inaccessible here (↗ see page 392)	You are attempting to reference a instance member from within a class procedure.
E2209: Field declarations not allowed in interface type (↗ see page 392)	An interface has been encountered which contains definitions of fields; this is not permitted.
x2044: Chmod error on '%s' (↗ see page 393)	The file permissions are not properly set on a file. See the chmod man page for more information.
x2043: Close error on '%s' (↗ see page 393)	The compiler encountered an error while closing an input or output file. This should rarely happen. If it does, the most likely cause is a full or bad disk.
F2039: Could not create output file '%s' (↗ see page 393)	The compiler could not create an output file. This can be a compiled unit file (.dcu), an executable file, a map file or an object file. Most likely causes are a nonexistent directory or a write protected file or disk.
x2141: Bad file format: '%s' (↗ see page 393)	The compiler state file has become corrupted. It is not possible to reload the previous compiler state. Delete the corrupt file.
E2288: File name too long (exceeds %d characters) (↗ see page 393)	A file path specified in the compiler options exceeds the compiler's file buffer length.
x1026: File not found: '%s' (↗ see page 393)	This error message occurs when the compiler cannot find an input file. This can be a source file, a compiled unit file (.dcuil file), an include, an object file or a resource file. Check the spelling of the name and the relevant search path.
F1027: Unit not found: '%s' or binary equivalents (%s) (↗ see page 394)	This error message occurs when the compiler cannot find a referenced unit (.dcuil) file. Check the spelling of the referenced file name and the relevant search path.
x2041: Read error on '%s' (↗ see page 394)	The compiler encountered a read error on an input file. This should never happen - if it does, the most likely cause is corrupt data.
F2040: Seek error on '%s' (↗ see page 394)	The compiler encountered a seek error on an input or output file. This should never happen - if it does, the most likely cause is corrupt data.
E2002: File type not allowed here (↗ see page 394)	File types are not allowed as value parameters and as the base type of a file type itself. They are also not allowed as function return types, and you cannot assign them - those errors will however produce a different error message.
x2042: Write error on '%s' (↗ see page 395)	The compiler encountered a write error while writing to an output file. Most likely, the output disk is full.

E2351: Final methods must be virtual or dynamic (↗ see page 395)	No further information is available for this error or warning.
E2155: Type '%s' needs finalization - not allowed in file type (↗ see page 395)	Certain types are treated specially by the compiler on an internal basis in that they must be correctly finalized to release any resources that they might currently own. Because the compiler cannot determine what type is actually stored in a record's variant section at runtime, it is not possible to guarantee that these special data types are correctly finalized.
E2154: Type '%s' needs finalization - not allowed in variant record (↗ see page 395)	Certain types are treated specially by the compiler on an internal basis in that they must be correctly finalized to release any resources that they might currently own. Because the compiler cannot determine what type is actually stored in a record's variant section at runtime, it is not possible to guarantee that these special data types are correctly finalized.
E2103: 16-Bit fixup encountered in object file '%s' (↗ see page 396)	A 16-bit fixup has been found in one of the object modules linked to your program with the $L compiler directive. The compiler only supports 32 bit fixups in linked object modules. Make sure that the linked object module is a 32 bit object module.
W1037: FOR-Loop variable '%s' may be undefined after loop (↗ see page 396)	This warning is issued if the value of a for loop control variable is used after the loop. You can only rely on the final value of a for loop control variable if the loop is left with a goto or exit statement. The purpose of this restriction is to enable the compiler to generate efficient code for the for loop.
W1015: FOR-Loop variable '%s' cannot be passed as var parameter (↗ see page 397)	An attempt has been made to pass the control variable of a FOR-loop to a procedure or function which takes a var parameter. This is a warning because the procedure which receives the control variable is able to modify it, thereby changing the semantics of the FOR-loop which issued the call.
E2032: For loop control variable must have ordinal type (↗ see page 398)	The control variable of a for loop must have type Boolean, Char, WideChar, Integer, an enumerated type, or a subrange type.
x1019: For loop control variable must be simple local variable (↗ see page 398)	This error message is given when the control variable of a for statement is not a simple variable (but a component of a record, for instance), or if it is not local to the procedure containing the for statement. For backward compatibility reasons, it is legal to use a global variable as the control variable - the compiler gives a warning in this case. Note that using a local variable will also generate more efficient code.
E2037: Declaration of '%s' differs from previous declaration (↗ see page 399)	This error message occurs when the declaration of a procedure, function, method, constructor or destructor differs from its previous (forward) declaration. This error message also occurs when you try to override a virtual method, but the overriding method has a different parameter list, calling convention etc.
E2065: Unsatisfied forward or external declaration: '%s' (↗ see page 400)	This error message appears when you have a forward or external declaration of a procedure or function, or a declaration of a method in a class or object type, and you don't define the procedure, function or method anywhere. Maybe the definition is really missing, or maybe its name is just misspelled. Note that a declaration of a procedure or function in the interface section of a unit is equivalent to a forward declaration - you have to supply the implementation (the body of the procedure or function) in the implementation section. Similarly, the declaration of a method in a... more (↗ see page 400)
W1011: Text after final 'END.' - ignored by compiler (↗ see page 401)	This warning is given when there is still source text after the final end and the period that constitute the logical end of the program. Possibly the nesting of begin-end is inconsistent (there is one end too many somewhere). Check whether you intended the source text to be ignored by the compiler - maybe it is actually quite important.
E2127: 'GOTO %s' leads into or out of TRY statement (↗ see page 401)	The GOTO statement cannot jump into or out of an exception handling statement.
E2295: A class helper cannot introduce a destructor (↗ see page 402)	Class helpers cannot declare destructors.

E2172: Necessary library helper function was eliminated by linker (%s) (see page 402)	The integrated debugger is attempting to use some of the compiler helper functions to perform the requested evaluate. The linker, on the other hand, determined that the helper function was not actually used by the program and it did not link it into the program. 1. Create a new application. 2. Place a button on the form. 3. Double click the button to be taken to the 'click' method. 4. Add a global variable, 'v', of type String to the interface section. 5. Add a global variable, 'p', of type PChar to the interface section. The click method should read as: 1. procedure TForm1.Button1Click(Sender: TObject); begin... more (see page 402)
W1010: Method '%s' hides virtual method of base type '%s' (see page 403)	You have declared a method which has the same name as a virtual method in the base class. Your new method is not a virtual method; it will hide access to the base's method of the same name.
W1009: Redeclaration of '%s' hides a member in the base class (see page 404)	A property has been created in a class with the same name of a variable contained in one of the base classes. One possible, and not altogether apparent, reason for getting this error is that a new version of the base class hierarchy has been installed and it contains new member variables which have names identical to your properties' names. -W
E2198: %s cannot be applied to a long string (see page 404)	It is not possible to use the standard function HIGH with long strings. The standard function HIGH can, however, be applied to old-style short strings. Since long strings dynamically size themselves, no analog to the HIGH function can be used. This error can be caused if you are porting a 16-bit application, in which case the only string type available was a short string. If this is the case, you can turn off the long strings with the $H command line switch or the long-form directive $LONGSTRINGS. If the HIGH was applied to a string parameter, but you still wish... more (see page 404)
W1034: $HPPEMIT '%s' ignored (see page 405)	The $HPPEMIT directive can only appear after the unit header.
x1008: Integer and HRESULT interchanged (see page 405)	In Delphi, Integer, Longint, and HRESULT are compatible types, but in C++ the types are not compatible and will produce differently mangled C++ parameter names. To ensure that there will not be problems linking object files created with the Delphi compiler this message alerts you to possible problems. If you are compiling your source to an object file, this is an error. Otherwise, it is a warning.
W1000: Symbol '%s' is deprecated (see page 406)	The symbol is tagged (using the **deprecated** hint directive) as no longer current and is maintained for compatibility only. You should consider updating your source code to use another symbol, if possible. The **$WARN** SYMBOL_DEPRECATED ON/OFF compiler directive turns on or off all warnings about the **deprecated** directive on symbols in the current unit.
E2372: Identifier expected (see page 406)	No further information is available for this error or warning.
W1003: Symbol '%s' is experimental (see page 406)	An "experimental" directive has been used on an identifier. "Experimental" indicates the presence of a class or unit which is incomplete or not fully tested.
W1001: Symbol '%s' is specific to a library (see page 406)	The symbol is tagged (using the **library** hint directive) as one that may not be available in all libraries. If you are likely to use different libraries, it may cause a problem. The **$WARN** SYMBOL_LIBRARY ON/OFF compiler directive turns on or off all warnings about the **library** directive on symbols in the current unit.
W1002: Symbol '%s' is specific to a platform (see page 407)	The symbol is tagged (using the **platform** hint directive) as one that may not be available on all platforms. If you are writing cross-platform applications, it may cause a problem. The **$WARN** SYMBOL_PLATFORM ON/OFF compiler directive turns on or off all warnings about the **platform** directive on symbols in the current unit.
E2004: Identifier redeclared: '%s' (see page 407)	The given identifier has already been declared in this scope - you are trying to reuse its name for something else.
E2003: Undeclared identifier: '%s' (see page 407)	The compiler could not find the given identifier - most likely it has been misspelled either at the point of declaration or the point of use. It might be from another unit that has not mentioned a uses clause.

E2427: Only one of IID or GuidAttribute can be specified (↗ see page 407)	The GUID or IID of your interface can be specified using square brackets at the top of the interface declaration or using a .NET attribute before the interface declaration. You may use either style of GUID or IID declaration, but not both styles in the same type.
E2038: Illegal character in input file: '%s' (%s) (↗ see page 408)	The compiler found a character that is illegal in Delphi programs. This error message is caused most often by errors with string constants or comments.
E2182: '%s' clause not allowed in OLE automation section (↗ see page 408)	INDEX, STORED, DEFAULT and NODEFAULT are not allowed in OLE automation sections.
E2231: '%s' directive not allowed in dispinterface type (↗ see page 409)	You have specified a clause in a dispinterface type which is not allowed.
E2207: '%s' clause not allowed in interface type (↗ see page 409)	The clause noted in the message is not allowed in an interface type. Typically this error indicates that an illegal directive has been specified for a property field in the interface.
E2176: Illegal type in OLE automation section: '%s' (↗ see page 410)	<typename> is not an allowed type in an OLE automation section. Only a small subset of all the valid Delphi language types are allowed in automation sections.
E2185: Overriding automated virtual method '%s' cannot specify a dispid (↗ see page 411)	The dispid declared for the original virtual automated procedure declaration must be used by all overriding procedures in derived classes.
E2068: Illegal reference to symbol '%s' in object file '%s' (↗ see page 412)	This error message is given if an object file loaded with a $L or $LINK directive contains a reference to a Delphi symbol that is not a procedure, function, variable, typed constant or thread local variable.
E2139: Illegal message method index (↗ see page 412)	You have specified value for your message index which <= 0.
E2224: $DESIGNONLY and $RUNONLY only allowed in package unit (↗ see page 412)	The compiler has encountered either $designonly or $runonly in a source file which is not a package. These directives affect the way that the IDE will treat a package file, and therefore can only be contained in package source files.
E2184: %s section valid only in class types (↗ see page 412)	Interfaces and records may not contain published sections. Records may not contain protected sections.
W1043: Imagebase $%X is not a multiple of 64k. Rounding down to $%X (↗ see page 413)	You can set an imagebase for a DLL to position it in a specific location in memory using the **$IMAGEBASE** compiler directive. The **$IMAGEBASE** directive controls the default load address for an application, DLL, or package. The number specified as the imagebase in the directive must be a multiple of 64K (that is, a hex number must have zeros as the last 4 digits), otherwise, it will be rounded down to the nearest multiple, and you will receive this compiler message.
E2227: Imagebase is too high - program exceeds 2 GB limit (↗ see page 413)	There are three ways to cause this error: 1. Specify a large enough imagebase that, when compiled, the application code passes the 2GB boundary. 2. Specify an imagebase via the command line which is above 2GB. 3. Specify an imagebase via $imagebase which is above 2GB. The only solution to this problem is to lower the imagebase address sufficiently so that the entire application will fit below the 2GB limit.
E2260: Implements clause not allowed together with index clause (↗ see page 413)	You have tried to use an index clause with an implements clause. Index specifiers allow several properties to share the same access method while representing different values. The implements directive allows you to delegate implementation of an interface to a property in the implementing class but it cannot take an index specifier.
E2263: Implements getter cannot be dynamic or message method (↗ see page 413)	An attempt has been made to use a dynamic or message method as a property accessor of a property which has an implements clause.
E2264: Cannot have method resolutions for interface '%s' (↗ see page 414)	An attempt has been made to use a method resolution clause for an interface named in an implements clause.
E2258: Implements clause only allowed within class types (↗ see page 415)	The interface definition in this example attempts to use an implements clause which causes the error.
E2259: Implements clause only allowed for properties of class or interface type (↗ see page 415)	An attempt has been made to use the implements clause with an improper type. Only class or interface types may be used.
E2262: Implements getter must be %s calling convention (↗ see page 415)	The compiler has encountered a getter or setter which does not have the correct calling convention.
E2265: Interface '%s' not mentioned in interface list (↗ see page 416)	An implements clause references an interface which is not mentioned in the interface list of the class.
E2261: Implements clause only allowed for readable property (↗ see page 416)	The compiler has encountered a "write only" property that claims to implement an interface. A property must be read/write to use the implements clause.
x1033: Unit '%s' implicitly imported into package '%s' (↗ see page 417)	The unit specified was not named in the contains clause of the package, but a unit which has already been included in the package imports it. This message will help the programmer avoid violating the rule that a unit may not reside in more than one related package. Ignoring the warning, will cause the unit to be put into the package. You could also explicitly list the named unit in the contains clause of the package to accomplish the same result and avoid the warning altogether. Or, you could alter the package list to load the named unit from another... more (↗ see page 417)
W1040: Implicit use of Variants unit (↗ see page 417)	If your application is using a Variant type, the compiler includes the Variant unit in the uses clause but warns you that you should add it explicitly.

E2420: Interface '%s' used in '%s' is not yet completely defined (↗ see page 417)	Interface used in is not yet completely defined. Forward declared interfaces must be declared in the same type section that they are used in. As an example the following code will not compile because of the above error message:
E2086: Type '%s' is not yet completely defined (↗ see page 418)	This error occurs if there is either a reference to a type that is just being defined, or if there is a forward declared class type in a type section and no final declaration of that type.
E2195: Cannot initialize local variables (↗ see page 418)	The compiler disallows the use of initialized local variables.
E2196: Cannot initialize multiple variables (↗ see page 419)	Variable initialization can only occur when variables are declared individually.
E2194: Cannot initialize thread local variables (↗ see page 419)	The compiler does not allow initialization of thread local variables.
E2072: Number of elements (%d) differs from declaration (%d) (↗ see page 420)	This error message appears when you declare a typed constant or initialized variable of array type, but do not supply the appropriate number of elements.
E2428: Field '%s' needs initialization - not allowed in CLS compliant value types (↗ see page 420)	CLS-compliant value types cannot have fields that require initialization. See ECMA 335, Partition II, Section 12.
E2418: Type '%s' needs initialization - not allowed in variant record (↗ see page 420)	Type needs initialization - not allowed in variant record. Variant records do not allow types that need initialization in their variant field list since each variant field references the same memory location. As an example, the following code will not compile because the array type needs to be initialized.
E2426: Inline function must not have asm block (↗ see page 421)	Inline functions can not include an asm block. To avoid this error, remove the inline directive from your function or use Pascal code to express the statements in the asm block.
E2442: Inline directive not allowed in constructor or destructor (↗ see page 421)	Remove the inline directive to prevent this error.
H2444: Inline function '%s' has not been expanded because accessing member '%s' is inaccessible (↗ see page 421)	An inline function cannot be expanded when the inline function body refers to a restricted member that is not accessible where the function is called. For example, if an inline function refers to a **strict private** field and this function is called from outside the class (e.g. from a global procedure), the field is not accessible at the call site and the inline function is not expanded.
E2425: Inline methods must not be virtual nor dynamic (↗ see page 421)	In order for an inline method to be inserted inline at compile-time, the method must be bound at compile-time. Virtual and dynamic methods are not bound until run-time, so they cannot be inserted inline. Make sure your method is static if you wish it to be inline.
E2449: Inlined nested routine '%s' cannot access outer scope variable '%s' (↗ see page 421)	You can use the **inline** directive with nested procedures and functions. However, a nested procedure or function that refers to a variable that is local to the outer procedure is not eligible for inlining.
H2445: Inline function '%s' has not been expanded because its unit '%s' is specified in USES statement of IMPLEMENTATION section and current function is inline function or being inline function (↗ see page 421)	Inline functions are not expanded between circularly dependent units.
H2443: Inline function '%s' has not been expanded because unit '%s' is not specified in USES list (↗ see page 421)	This situation may occur if an inline function refers to a type in a unit that is not explicitly used by the function's unit. For example, this may happen if the function uses **inherited** to refer to methods inherited from a distant ancestor, and that ancestor's unit is not explicitly specified in the uses list of the function's unit. If the inline function's code is to be expanded, then the unit that calls the function must explicitly use the unit where the ancestor type is exposed.
E2441: Inline function declared in interface section must not use local symbol '%s' (↗ see page 422)	This error occurs when an inline function is declared in the interface section and it refers to a symbol that is not visible outside the unit. Expanding the inline function in another unit would require accessing the local symbol from outside the unit, which is not permitted. To correct this error, move the local symbol declaration to the interface section, or make it an instance variable or class variable of the function's class type.
E2382: Cannot call constructors using instance variables (↗ see page 422)	No further information is available for this error or warning.
E2102: Integer constant too large (↗ see page 422)	You have specified an integer constant that requires more than 64 bits to represent.
F2084: Internal Error: %s%d (↗ see page 422)	Occasionally when compiling an application in Delphi, the compile will halt and display an error message that reads, for example:
E2232: Interface '%s' has no interface identification (↗ see page 423)	You have attempted to assign an interface to a GUID type, but the interface was not defined with a GUID.
E2291: Missing implementation of interface method %s.%s (↗ see page 423)	This indicates that you have forgotten to implement a method required by an interface supported by your class type.
E2211: Declaration of '%s' differs from declaration in interface '%s' (↗ see page 423)	A method declared in a class which implements an interface is different from the definition which appears in the interface. Probable causes are that a parameter type or return value is declared differently, the method appearing in the class is a message method, the identifier in the class is a field or the identifier in the class is a property, which does not match with the definition in the interface.
E2208: Interface '%s' already implemented by '%s' (↗ see page 425)	The class specified by name2 has specified the interface name1 more than once in the inheritance section of the class definition.

E2089: Invalid typecast (see page 425)	This error message is issued for type casts not allowed by the rules. The following kinds of casts are allowed: • Ordinal or pointer type to another ordinal or pointer type • A character, string, array of character or pchar to a string • An ordinal, real, string or variant to a variant • A variant to an ordinal, real, string or variant • A variable reference to any type of the same size. Note that casting real types to integer can be performed with the standard functions Trunc and Round. There are other transfer functions like Ord and Chr that might make your intention... more (see page 425)
E2424: Codepage '%s' is not installed on this machine (see page 426)	This message occurs if you specify a codepage using the `--codepage=nnn` command line switch and the codepage you specify is not available on the machine. See your operating system documentation for details on how to install codepages.
E2173: Missing or invalid conditional symbol in '$%s' directive (see page 426)	The $IFDEF, $IFNDEF, $DEFINE and $UNDEF directives require that a symbol follow them.
x1030: Invalid compiler directive: '%s' (see page 426)	This error message means there is an error in a compiler directive or in a command line option. Here are some possible error situations: • An external declaration was syntactically incorrect. • A command line option or an option in a DCC32.CFG file was not recognized by the compiler or was invalid. For example, '-$M100' is invalid because the minimum stack size must be at least 1024. • The compiler found a $XXXXX directive, but could not recognize it. It was probably misspelled. • The compiler found a $ELSE or $ENDIF directive, but no preceding $IFDEF, $IFNDEF or $IFOPT directive. • (*$IFOPT*) was not followed... more (see page 426)
E2298: read/write not allowed for CLR events. Use Include/Exclude procedure (see page 427)	Multicast events cannot be assigned to or read from like traditional Delphi read/write events. Use Include/Exclude to add or remove methods.
E2138: Invalid message parameter list (see page 427)	A message procedure can take only one, VAR, parameter; it's type is not checked.
E2294: A class helper that descends from '%s' can only help classes that are descendents '%s' (see page 428)	The object type specified in the "for" clause of a class helper declaration is not a descendent of the object type specified in the "for" clause of the class helper's ancestor type.
E2296: A constructor introduced in a class helper must call the parameterless constructor of the helped class as the first statement (see page 428)	The first statement in a class helper constructor must be "inherited Create;"
E2387: The key container name '%s' does not exist (see page 428)	No further information is available for this error or warning.
E2388: Unrecognized strong name key file '%s' (see page 428)	No further information is available for this error or warning.
E2432: %s cannot be applied to a rectangular dynamic array (see page 429)	This error may arise if you attempt to pass a dynamically allocated rectangular array to the Low or High function. If you receive this error, use a static or ragged (non-rectangular) array. See the Delphi Language Guide for details.
E2393: Invalid operator declaration (see page 429)	No further information is available for this error or warning.
E2174: '%s' not previously declared as a PROPERTY (see page 429)	You have attempted to hoist a property to a different visibility level by redeclaration, but <name> in the base class was not declared as a property. -W
E2376: STATIC can only be used on non-virtual class methods (see page 430)	No further information is available for this error or warning.
E2415: Could not import assembly '%s' because it contains namespace '%s' (see page 430)	The Borland.Delphi.System unit may only be loaded from the Borland.Delphi.dll assembly. This error will occur if Borland.Delphi.System is attempted to be loaded from an alternative assembly.
E2416: Could not import package '%s' because it contains system unit '%s' (see page 430)	The Borland.Delphi.System unit may only be loaded from the Borland.Delphi.dll package. This error will occur if Borland.Delphi.System is attempted to be loaded from an alternative package.

F2438: UCS-4 text encoding not supported. Convert to UCS-2 or UTF-8 (↗ see page 430)	This error is encountered when a source file has a UCS-4 encoding, as indicated by its Byte-Order-Mark (BOM). The compiler does not support compilation of source files in UCS-4 Unicode encoding. To solve this problem, convert the source file to UCS-2 or UTF-8 encoding.
E2386: Invalid version string '%s' specified in %s (↗ see page 430)	No further information is available for this error or warning.
E2120: LOOP/JCXZ distance out of range (↗ see page 430)	You have specified a LOOP or JCXZ destination which is out of range. You should not receive this error as the jump range is 2Gb for LOOP and JCXZ instructions.
E2049: Label declaration not allowed in interface part (↗ see page 430)	This error occurs when you declare a label in the interface part of a unit.
E2073: Label already defined: '%s' (↗ see page 431)	This error message is given when a label is set on more than one statement.
E2074: Label declared and referenced, but not set: '%s' (↗ see page 431)	You declared and used a label in your program, but the label definition was not encountered in the source code.
F2069: Line too long (more than 1023 characters) (↗ see page 432)	This error message is given when the length of a line in the source file exceeds 255 characters. Usually, you can divide the long line into two shorter lines. If you need a really long string constant, you can break it into several pieces on consecutive lines that you concatenate with the '+' operator.
E2364: Cross-assembly protected reference to [%s]%s.%s in %s.%s (↗ see page 432)	In Delphi for .NET, members with protected visibility cannot be accessed outside of the assembly in which they are defined. If possible, you may want to use the publicly-exposed members of the class to accomplish your goal. Other ways to resolve this error: • Increase the visibility of the member from protected to public, so it can be accessed outside of its assembly. • “Link in” the assembly where the protected member is defined, so that this assembly is incorporated into the assembly you are building, and the access will be inside the assembly.
W1053: Local PInvoke code has not been made because external routine '%s' in package '%s' is defined using package local types in its custom attributes (↗ see page 432)	This warning may arise when an external package uses PInvoke to access Win32 library code, and that package exposes the PInvoke definition through a public export. In these cases the compiler will attempt to link directly to the Win32 library by copying the PInvoke definition to the local assembly, rather than linking to the public export in the external package. This is more secure and can also improve runtime performance. This warning message is issued if the compiler is unable to emit the PInvoke definition locally, because the external assembly uses locally-defined types for a custom attribute. To avoid this... more (↗ see page 432)
E2094: Local procedure/function '%s' assigned to procedure variable (↗ see page 433)	This error message is issued if you try to assign a local procedure to a procedure variable, or pass it as a procedural parameter. This is illegal, because the local procedure could then be called even if the enclosing procedure is not active. This situation would cause the program to crash if the local procedure tried to access any variables of the enclosing procedure.
E2189: Thread local variables cannot be local to a function (↗ see page 434)	Thread local variables must be declared at a global scope.
W1042: Error converting locale string '%s' to Unicode. String truncated. Is your LANG environment variable set correctly? (↗ see page 434)	This message occurs when you are trying to convert strings to Unicode and the string contains characters that are not valid for the current locale. For example, this may occur when converting WideString to AnsiString or if attempting to display Japanese characters in an English locale.
E2011: Low bound exceeds high bound (↗ see page 434)	This error message is given when either the low bound of a subrange type is greater than the high bound, or the low bound of a case label range is greater than the high bound.
H2440: Inline method visibility is not lower or same visibility of accessing member '%s.%s' (↗ see page 435)	A member that is accessed within the body of an inline method must be accessible anywhere that the inline method is called. Therefore, the member must be at least as visible as the inline method. Here is an example of code that will raise this error:
E2204: Improper GUID syntax (↗ see page 435)	The GUID encountered in the program source is malformed. A GUID must be of the form: 00000000-0000-0000-0000-000000000000.
E2348: Metadata - Bad input parameters (↗ see page 435)	No further information is available for this error or warning.
E2347: Metadata - Bad binary signature (↗ see page 436)	No further information is available for this error or warning.
E2349: Metadata - Cannot resolve typeref (↗ see page 436)	No further information is available for this error or warning.
E2345: Metadata - Attempt to define an object that already exists (↗ see page 436)	No further information is available for this error or warning.
E2346: Metadata - A guid was not provided where one was required (↗ see page 436)	No further information is available for this error or warning.

E2350: Metadata - No logical space left to create more user strings (see page 436)	No further information is available for this error or warning.
F2046: Out of memory (see page 436)	The compiler ran out of memory. This should rarely happen. If it does, make sure your swap file is large enough and that there is still room on the disk.
x1054: Linker error: %s (see page 436)	This message emits a warning or other text generated using the $MESSAGE directive.
E2096: Method identifier expected (see page 436)	This error message will be issued in several different situations: • Properties in an automated section must use methods for access, they cannot use fields in their read or write clauses. • You tried to call a class method with the "ClassType.MethodName" syntax, but "MethodName" was not the name of a method. • You tried calling an inherited with the "Inherited MethodName" syntax, but "MethodName" was not the name of a method.
E2433: Method declarations not allowed in anonymous record or local record type (see page 437)	Record types that are declared in local scopes or declared in-place in variable declarations can only contain field declarations. For advanced features in record types (such as methods, properties, and nested types), the record type must be an explicitly declared global type.
E2234: Getter or setter for property '%s' cannot be found (see page 437)	During translation of a unit to a C++ header file, the compiler is unable to locate a named symbol which is to be used as a getter or setter for a property. This is usually caused by having nested records in the class and the accessor is a field in the nested record.
E2095: Missing ENDIF directive (see page 437)	This error message is issued if the compiler does not find a corresponding $ENDIF directive after an $IFDEF, $IFNDEF or $IFOPT directive.
E2403: Add or remove accessor for event '%s' cannot be found (see page 438)	No further information is available for this error or warning.
E2253: Ancestor type '%s' does not have an accessible default constructor (see page 438)	The ancestor of the class being compiled does not have an accessible default constructor. This error only occurs with the byte code version of the compiler.
E2066: Missing operator or semicolon (see page 438)	This error message appears if there is no operator between two subexpressions, or no semicolon between two statements. Often, a semicolon is missing on the previous line.
E2202: Required package '%s' not found (see page 439)	The package, which is referenced in the message, appears on the package list, either explicitly or through a requires clause of another unit appearing on the package list, but cannot be found by the compiler. The solution to this problem is to ensure that the DCP file for the named package is in one of the units named in the library path.
E2035: Not enough actual parameters (see page 439)	This error message occurs when a call to procedure or function gives less parameters than specified in the procedure or function declaration. This can also occur for calls to standard procedures or functions.
E2067: Missing parameter type (see page 439)	This error message is issued when a parameter list gives no type for a value parameter. Leaving off the type is legal for constant and variable parameters.
E2151: Could not load RLINK32.DLL (see page 440)	RLINK32 could not be found. Please ensure that it is on the path. Contact CodeGear if you encounter this error.
E2404: Cannot mix READ/WRITE property accessors with ADD/REMOVE accessors (see page 440)	No further information is available for this error or warning.
E2359: Multiple class constructors in class %s: %s and %s (see page 440)	No further information is available for this error or warning.
E2287: Cannot export '%s' multiple times (see page 440)	This message is not used in this product.
E2085: Unit name mismatch: '%s' '%s' (see page 440)	The unit name in the top unit is case sensitive and must match the name with respect to upper- and lowercase letters exactly. The unit name is case sensitive only in the unit declaration.
E2016: Array type required (see page 440)	This error message is given if you either index into an operand that is not an array, or if you pass an argument that is not an array to an open array parameter.
E2012: Type of expression must be BOOLEAN (see page 441)	This error message is output when an expression serves as a condition and must therefore be of Boolean type. This is the case for the controlling expression of the if, while and repeat statements, and for the expression that controls a conditional breakpoint.

E2021: Class type required (↗ see page 441)	In certain situations the compiler requires a class type: • As the ancestor of a class type • In the on-clause of a try-except statement • As the first argument of a raise statement • As the final type of a forward declared class type
E2076: This form of method call only allowed for class methods (↗ see page 441)	You were trying to call a normal method by just supplying the class type, not an actual instance. This is only allowed for class methods and constructors, not normal methods and destructors.
E2149: Class does not have a default property (↗ see page 442)	You have used a class instance variable in an array expression, but the class type has not declared a default array property.
E2168: Field or method identifier expected (↗ see page 443)	You have specified an identifier for a read or write clause to a property which is not a field or method.
E2022: Class helper type required (↗ see page 443)	When declaring a class helper type with an ancestor clause, the ancestor type must be a class helper.
E2380: Instance or class static method expected (↗ see page 444)	No further information is available for this error or warning.
E2013: Type of expression must be INTEGER (↗ see page 444)	This error message is only given when the constant expression that specifies the number of characters in a string type is not of type integer.
E2205: Interface type required (↗ see page 444)	A type, which is an interface, was expected but not found. A common cause of this error is the specification of a user-defined type that has not been declared as an interface type.
E2031: Label expected (↗ see page 445)	This error message occurs if the identifier given in a goto statement or used as a label in inline assembly is not declared as a label.
E2075: This form of method call only allowed in methods of derived types (↗ see page 445)	This error message is issued if you try to make a call to a method of an ancestor type, but you are in fact not in a method.
E2019: Object type required (↗ see page 446)	This error is given whenever an object type is expected by the compiler. For instance, the ancestor type of an object must also be an object type.
E2020: Object or class type required (↗ see page 446)	This error message is given when the syntax 'Typename.Methodname' is used, but the typename does not refer to an object or class type.
E2254: Overloaded procedure '%s' must be marked with the 'overload' directive (↗ see page 447)	The compiler has encountered a procedure, which is not marked overload, with the same name as a procedure already marked overload. All overloaded procedures must be marked as such.
E2017: Pointer type required (↗ see page 447)	This error message is given when you apply the dereferencing operator '^' to an operand that is not a pointer, and, as a very special case, when the second operand in a 'Raise <exception> at <address>' statement is not a pointer.
E2267: Previous declaration of '%s' was not marked with the 'overload' directive (↗ see page 448)	There are two solutions to this problem. You can either remove the attempt at overloading or you can mark the original declaration with the overload directive. The example shown here marks the original declaration.
E2121: Procedure or function name expected (↗ see page 448)	You have specified an identifier which does not represent a procedure or function in an EXPORTS clause.
E2299: Property required (↗ see page 449)	You need to add a property to your program. The declaration of a property specifies a name and a type, and includes at least one access specifier. The syntax of a property declaration is:
E2018: Record, object or class type required (↗ see page 449)	The compiler was expecting to find the type name which specified a record, object or class but did not find one.
E2023: Function needs result type (↗ see page 450)	You have declared a function, but have not specified a return type.
E2366: Global procedure or class static method expected (↗ see page 450)	No further information is available for this error or warning.
E2036: Variable required (↗ see page 451)	This error message occurs when you try to take the address of an expression or a constant.
E2082: TYPEOF can only be applied to object types with a VMT (↗ see page 451)	This error message is issued if you try to apply the standard function TypeOf to an object type that does not have a virtual method table. A simple workaround is to declare a dummy virtual procedure to force the compiler to generate a VMT.
E2014: Statement expected, but expression of type '%s' found (↗ see page 452)	The compiler was expecting to find a statement, but instead it found an expression of the specified type.
E2279: Too many nested conditional directives (↗ see page 452)	Conditional-directive constructions can be nested up to 32 levels deep.
E2409: Fully qualified nested type name %s exceeds 1024 byte limit (↗ see page 452)	No further information is available for this error or warning.
E2079: Procedure NEW needs constructor (↗ see page 452)	This error message is issued when an identifier given in the parameter list to New is not a constructor.

W1039: No configuration files found (↗ see page 453)	The compiler could not locate the configuration files referred to in the source code.
E2256: Dispose not supported (nor necessary) for dynamic arrays (↗ see page 453)	The compiler has encountered a use of the standard procedure DISPOSE on a dynamic array. Dynamic arrays are reference counted and will automatically free themselves when there are no longer any references to them.
E2250: There is no overloaded version of '%s' that can be called with these arguments (↗ see page 454)	An attempt has been made to call an overloaded function that cannot be resolved with the current set of overloads.
E2450: There is no overloaded version of array property '%s' that can be used with these arguments (↗ see page 454)	To correct this error, either change the arguments so that their types match a version of the array property, or add a new overload of the array property with types that match the arguments.
E2273: No overloaded version of '%s' with this parameter list exists (↗ see page 455)	An attempt has been made to call an overloaded procedure but no suitable match could be found.
E2025: Procedure cannot have a result type (↗ see page 455)	You have declared a procedure, but given it a result type. Either you really meant to declare a function, or you should delete the result type.
W1035: Return value of function '%s' might be undefined (↗ see page 456)	This warning is displayed if the return value of a function has not been assigned a value on every code path. To put it another way, the function could execute so that it never assigns anything to the return value.
E2134: Type '%s' has no type info (↗ see page 457)	You have applied the TypeInfo standard procedure to a type identifier which does not have any run-time type information associated with it.
E2220: Never-build package '%s' requires always-build package '%s' (↗ see page 458)	You are attempting to create a no-build package which requires an always-build package. Since the interface of an always-build package can change at anytime, and since giving the no-build flag instructs the compiler to assume that a package is up-to-date, each no-build package can only require other packages that are also marked no-build.
E2093: Label '%s' is not declared in current procedure (↗ see page 458)	In contrast to standard Pascal, Borland's Delphi language does not allow a goto to jump out of the current procedure. However, his construct is mainly useful for error handling, and the Delphi language provides a more general and structured mechanism to deal with errors: exception handling.
x2269: Overriding virtual method '%s.%s' has lower visibility (%s) than base class '%s' (%s) (↗ see page 459)	The method named in the error message has been declared as an override of a virtual method in a base class, but the visibility in the current class is lower than that used in the base class for the same method. While the visibility rules of Delphil would seem to indicate that the function cannot be seen, the rules of invoking virtual functions will cause the function to be properly invoked through a virtual call. Generally, this means that the method of the derived class was declared in a private or protected section while the method of the base class... more (↗ see page 459)
E2411: Unit %s in package %s refers to unit %s which is not found in any package. Packaged units must refer only to packaged units (↗ see page 460)	No further information is available for this error or warning.
E2236: Constructors and destructors must have %s calling convention (↗ see page 460)	An attempt has been made to change the calling convention of a constructor or destructor from the default calling convention.
E2179: Only register calling convention allowed in OLE automation section (↗ see page 461)	You have specified an illegal calling convention on a method appearing in an 'automated' section of a class declaration.
E2270: Published property getters and setters must have %s calling convention (↗ see page 462)	A property appearing in a published section has a getter or setter procedure that does not have the correct calling convention.
E2391: Potentially polymorphic constructor calls must be virtual (↗ see page 462)	No further information is available for this error or warning.
E2242: '%s' is not the name of a unit (↗ see page 462)	The $NOINCLUDE directive must be given a known unit name.
E2064: Left side cannot be assigned to (↗ see page 462)	This error message is given when you try to modify a read-only object like a constant, a constant parameter, or the return value of function.
E2430: for-in statement cannot operate on collection type '%s' (↗ see page 463)	A for-in statement can only operate on the following collection types: • Primitive types that the compiler recognizes, such as arrays, sets or strings • Types that implement IEnumerable • Types that implement the GetEnumerator pattern as documented in the Delphi Language Guide Ensure that the specified type meets these requirements.
H2135: FOR or WHILE loop executes zero times - deleted (↗ see page 463)	The compiler has determined that the specified looping structure will not ever execute, so as an optimization it will remove it. Example:
E2248: Cannot use old style object types when compiling to byte code (↗ see page 464)	Old-style Object types are illegal when compiling to byte code.

E2058: Class, interface and object types only allowed in type section (↗ see page 464)	Class or object types must always be declared with an explicit type declaration in a type section - unlike record types, they cannot be anonymous. The main reason for this is that there would be no way you could declare the methods of that type - after all, there is no type name.
E2059: Local class, interface or object types not allowed (↗ see page 465)	Class and object cannot be declared local to a procedure.
E2062: Virtual constructors are not allowed (↗ see page 465)	Unlike class types, object types can only have static constructors.
E2439: Inline function must not have open array argument (↗ see page 466)	To avoid this error, remove the **inline** directive or use an explicitly-declared dynamic array type instead of an open array argument.
W1049: value '%s' for option %s was truncated (↗ see page 466)	String based compiler options such as unit search paths have finite buffer limits. This message indicates you have exceeded the buffer limit.
E2001: Ordinal type required (↗ see page 466)	The compiler required an ordinal type at this point. Ordinal types are the predefined types Integer, Char, WideChar, Boolean, and declared enumerated types. Ordinal types are required in several different situations: • The index type of an array must be ordinal. • The low and high bounds of a subrange type must be constant expressions of ordinal type. • The element type of a set must be an ordinal type. • The selection expression of a case statement must be of ordinal type. • The first argument to the standard procedures Inc and Dec must be a variable of either ordinal or pointer type.
E2271: Property getters and setters cannot be overloaded (↗ see page 467)	A property has specified an overloaded procedure as either its getter or setter.
H2365: Override method %s.%s should match case of ancestor %s.%s (↗ see page 468)	No further information is available for this error or warning.
E2137: Method '%s' not found in base class (↗ see page 468)	You have applied the 'override' directive to a method, but the compiler is unable to find a procedure of the same name in the base class.
E2352: Cannot override a final method (↗ see page 469)	No further information is available for this error or warning.
E2170: Cannot override a non-virtual method (↗ see page 469)	You have tried, in a derived class, to override a base method which was not declared as one of the virtual types.
F2220: Could not compile package '%s' (↗ see page 470)	An error occurred while trying to compile the package named in the message. The only solution to the problem is to correct the error and recompile the package.
E2199: Packages '%s' and '%s' both contain unit '%s' (↗ see page 470)	The project you are trying to compile is using two packages which both contain the same unit. It is illegal to have two packages which are used in the same project containing the same unit since this would cause an ambiguity for the compiler. A main cause of this problem is a poorly defined package set. The only solution to this problem is to redesign your package hierarchy to remove the ambiguity.
E2200: Package '%s' already contains unit '%s' (↗ see page 470)	The package you are compiling requires (either through the requires clause or the package list) another package which already contains the unit specified in the message. It is an error to have to related packages contain the same unit. The solution to this problem is to remove the unit from one of the packages or to remove the relation between the two packages.
W1031: Package '%s' will not be written to disk because -J option is enabled (↗ see page 470)	The compiler can't write the package to disk because the -J option is attempting to create an object file.
E2225: Never-build package '%s' must be recompiled (↗ see page 470)	The package referenced in the message was compiled as a never-build package, but it requires another package to which interface changes have been made. The named package cannot be used without recompiling because it was linked with a different interface of the required package. The only solution to this error is to manually recompile the offending package. Be sure to specify the never-build switch, if it is still desired.
H2235: Package '%s' does not use or export '%s.%s' (↗ see page 470)	You have compiled a unit into a package which contains a symbol which does not appear in the interface section of the unit, nor is it referenced by any code in the unit. In effect, this code is dead code and could be removed from the unit without changing the semantics of your program.
E2201: Need imported data reference ($G) to access '%s' from unit '%s' (↗ see page 470)	The unit named in the message was not compiled with the $G switch turned on.

W1032: Exported package threadvar '%s.%s' cannot be used outside of this package (see page 471)	Windows does not support the exporting of threadvar variables from a DLL, but since using packages is meant to be semantically equivalent to compiling a project without them, the Delphi compiler must somehow attempt to support this construct. This warning is to notify you that you have included a unit which contains a threadvar in an interface into a package. While this is not illegal, you will not be able to access the variable from a unit outside the package. Attempting to access this variable may appear to succeed, but it actually did not. A solution to this warning is... more (see page 471)
E2213: Bad packaged unit format: %s.%s (see page 471)	When the compiler attempted to load the specified unit from the package, it was found to be corrupt. This problem could be caused by an abnormal termination of the compiler when writing the package file (for example, a power loss). The first recommended action is to delete the offending DCP file and recompile the package.
E2006: PACKED not allowed here (see page 471)	The packed keyword is only legal for set, array, record, object, class and file types. In contrast to the 16-bit version of Delphi, packed will affect the layout of record, object and class types.
E2394: Parameterless constructors not allowed on record types (see page 472)	No further information is available for this error or warning.
E2363: Only methods of descendent types may access protected symbol [%s]%s.%s across assembly boundaries (see page 472)	No further information is available for this error or warning.
E2375: PRIVATE or PROTECTED expected (see page 472)	No further information is available for this error or warning.
W1045: Property declaration references ancestor private '%s.%s' (see page 472)	This warning indicates that your code is not portable to C++. This is important for component writers who plan to distribute custom components. In the Delphi language, you can declare a base class with a private member, and a child class in the same unit can refer to the private member. In C++, this construction is not permitted. To fix it, change the child to refer to either a protected member of the base class or a protected member of the child class. Following is an example of code that would cause this error:
H2219: Private symbol '%s' declared but never used (see page 472)	The symbol referenced appears in a private section of a class, but is never used by the class. It would be more memory efficient if you removed the unused private field from your class definition.
E2357: PROCEDURE, FUNCTION, or CONSTRUCTOR expected (see page 473)	No further information is available for this error or warning.
E2122: PROCEDURE or FUNCTION expected (see page 473)	This error message is produced by two different constructs, but in both cases the compiler is expecting to find the keyword 'procedure' or the keyword 'function'.
x2367: Case of property accessor method %s.%s should be %s.%s (see page 474)	No further information is available for this error or warning.
E2300: Cannot generate property accessor '%s' because '%s' already exists (see page 474)	No further information is available for this error or warning.
E2370: Cannot use inherited methods for interface property accessors (see page 474)	No further information is available for this error or warning.
H2369: Property accessor %s should be %s (see page 474)	No further information is available for this error or warning.
H2368: Visibility of property accessor method %s should match property %s.%s (see page 474)	No further information is available for this error or warning.
E2181: Redeclaration of property not allowed in OLE automation section (see page 474)	It is not allowed to move the visibility of a property into an automated section.
E2206: Property overrides not allowed in interface type (see page 475)	A property which was declared in a base interface has been overridden in an interface extension.
E2356: Property accessor must be an instance field or method (see page 476)	No further information is available for this error or warning.
E2434: Property declarations not allowed in anonymous record or local record type (see page 476)	Record types that are declared in local scopes or declared in-place in variable declarations can only contain field declarations. For advanced features in record types (such as methods, properties, and nested types), the record type must be an explicitly declared global type.
E2148: Dynamic method or message handler not allowed here (see page 476)	Dynamic and message methods cannot be used as accessor functions for properties.
E2233: Property '%s' inaccessible here (see page 477)	An attempt has been made to access a property through a class reference type. It is not possible to access fields nor properties of a class through a class reference.
E2275: property attribute 'label' cannot be an empty string (see page 477)	The error is output because the label attribute for g is an empty string.
E2292: '%s' must reference a property or field of class '%s' (see page 478)	In custom attribute declaration syntax, you can pass values to the constructor of the attribute class, followed by name=value pairs, where name is a property or field of the attribute class.
E2129: Cannot assign to a read-only property (see page 478)	The property to which you are attempting to assign a value did not specify a 'write' clause, thereby causing it to be a read-only property.

E2130: Cannot read a write-only property (↗ see page 479)	The property from which you are attempting to read a value did not specify a 'read' clause, thereby causing it to be a write-only property.
E2362: Cannot access protected symbol %s.%s (↗ see page 480)	No further information is available for this error or warning.
E2389: Protected member '%s' is inaccessible here (↗ see page 480)	No further information is available for this error or warning.
H2244: Pointer expression needs no Initialize/Finalize - need ^ operator? (↗ see page 480)	You have attempted to finalize a Pointer type.
E2186: Published Real property '%s' must be Single, Real, Double or Extended (↗ see page 480)	You have attempted to publish a property of type Real, which is not allowed. Published floating point properties must be Single, Double, or Extended.
E2187: Size of published set '%s' is >4 bytes (↗ see page 481)	The compiler does not allow sets greater than 32 bits to be contained in a published section. The size, in bytes, of a set can be calculated by High(setname) div 8 - Low(setname) div 8 + 1. -$M+
E2361: Cannot access private symbol %s.%s (↗ see page 481)	No further information is available for this error or warning.
E2390: Class must be sealed to call a private constructor without a type qualifier (↗ see page 481)	No further information is available for this error or warning.
E2398: Class methods in record types must be static (↗ see page 482)	No further information is available for this error or warning.
E2083: Order of fields in record constant differs from declaration (↗ see page 482)	This error message occurs if record fields in a typed constant or initialized variable are not initialized in declaration order.
E2419: Record type too large: exceeds 1 MB (↗ see page 482)	Records are limited to a size of 1MB according to the .NET SDK Documentation. Refer to Partition II Medatada 21.8 ClassLayout: 0x0F
E2245: Recursive include file %s (↗ see page 482)	The $I directive has been used to recursively include another file. You must check to make sure that all include files terminate without having cycles in them.
F2092: Program or unit '%s' recursively uses itself (↗ see page 482)	An attempt has been made for a unit to use itself.
E2214: Package '%s' is recursively required (↗ see page 483)	When compiling a package, the compiler determined that the package requires itself.
E2145: Re-raising an exception only allowed in exception handler (↗ see page 483)	You have used the syntax of the raise statement which is used to reraise an exception, but the compiler has determined that this reraise has occurred outside of an exception handler block. A limitation of the current exception handling mechanism disallows reraising exceptions from nested exception handlers. for the exception.
E2377: Unable to locate Borland.Delphi.Compiler.ResCvt.dll (↗ see page 484)	No further information is available for this error or warning.
E2381: Resource string length exceeds Windows limit of 4096 characters (↗ see page 484)	No further information is available for this error or warning.
E2024: Invalid function result type (↗ see page 484)	File types are not allowed as function result types.
Linker error: %s (↗ see page 484)	The resource linker (RLINK32) has encountered an error while processing a resource file. This error may be caused by any of the following reasons: • You have used a duplicate resource name. Rename one of the resources. • You have a corrupted resource file. You need to replace it with another version that is not corrupted or remove it. • You are using an unsupported resource type, such as a 16-bit resource or form template. • If converting resources such as 16-bit icons to 32-bit, the resource linker may have encountered problems.
Linker error: %s: %s (↗ see page 485)	The resource linker (RLINK32) has encountered an error while processing a resource file. A resource linked into the project has the same type and name, or same type and resource ID, as another resource linked into the project. (In Delphi, duplicate resources are ignored with a warning. In Kylix, duplicates cause an error.)
E2215: 16-Bit segment encountered in object file '%s' (↗ see page 485)	A 16-bit segment has been found in an object file that was loaded using the $L directive.
E2091: Segment/Offset pairs not supported in CodeGear 32-bit Pascal (↗ see page 485)	32-bit code no longer uses the segment/offset addressing scheme that 16-bit code used. In 16-bit versions of CodeGear Pascal, segment/offset pairs were used to declare absolute variables, and as arguments to the Ptr standard function. Note that absolute addresses should not be used in 32-bit protected mode programs. Instead appropriate Win32 API functions should be called.
E2153: ';' not allowed before 'ELSE' (↗ see page 485)	You have placed a ';' directly before an ELSE in an IF-ELSE statement. The reason for this is that the ';' is treated as a statement separator, not a statement terminator - IF-ELSE is one statement, a ';' cannot appear in the middle (unless you use compound statements).

E2028: Sets may have at most 256 elements (see page 486)	This error message appears when you try to declare a set type of more than 256 elements. More precisely, the ordinal values of the upper and lower bounds of the base type must be within the range 0..255.
E2282: Property setters cannot take var parameters (see page 486)	This message is displayed when you try to use a var parameter in a property setter parameter. The parameter of a property setter procedure cannot be a var or out parameter.
E2193: Slice standard function only allowed as open array argument (see page 487)	An attempt has been made to pass an array slice to a fixed size array. Array slices can only be sent to open array parameters. none
E2454: Slice standard function not allowed for VAR nor OUT argument (see page 487)	You cannot write back to a slice of an array, so you cannot use the slice standard function to pass an argument that is **var** or **out**. If you must modify the array, either pass in the full array or use an array variable to hold the desired part of the full array.
E2240: $EXTERNALSYM and $NODEFINE not allowed for '%s'; only global symbols (see page 487)	The $EXTERNALSYM and $NODEFINE directives can only be applied to global symbols.
W1014: String constant truncated to fit STRING[%ld] (see page 487)	A string constant is being assigned to a variable which is not large enough to contain the entire string. The compiler is alerting you to the fact that it is truncating the literal to fit into the variable. -W
E2354: String element cannot be passed to var parameter (see page 488)	No further information is available for this error or warning.
E2056: String literals may have at most 255 elements (see page 488)	This error message occurs when you declare a string type with more than 255 elements, if you assign a string literal of more than 255 characters to a variable of type ShortString, or when you have more than 255 characters in a single character string. Note that you can construct long string literals spanning more than one line by using the '+' operator to concatenate several string literals.
E2408: Can't extract strong name key from assembly %s (see page 489)	No further information is available for this error or warning.
W1044: Suspicious typecast of %s to %s (see page 489)	This warning flags typecasts like PWideChar(String) or PChar(WideString) which are casting between different string types without character conversion.
E2272: Cannot use reserved unit name '%s' (see page 489)	An attempt has been made to use one of the reserved unit names, such as System, as the name of a user-created unit. The names in the following list are currently reserved by the compiler. • System • SysInit
E2156: Expression too complicated (see page 489)	The compiler has encounter an expression in your source code that is too complicated for it to handle. Reduce the complexity of your expression by introducing some temporary variables.
E2283: Too many local constants. Use shorter procedures (see page 489)	One or more of your procedures contain so many string constant expressions that they exceed the compiler's internal storage limit. This can occur in code that is automatically generated. To fix this, you can shorten your procedures or declare contant identifiers instead of using so many literals in the code.
E2163: Too many conditional symbols (see page 490)	You have exceeded the memory allocated to conditional symbols defined on the command line (including configuration files). There are 256 bytes allocated for all the conditional symbols. Each conditional symbol requires 1 extra byte when stored in conditional symbol area. The only solution is to reduce the number of conditional compilation symbols contained on the command line (or in configuration files).
E2226: Compilation terminated; too many errors (see page 490)	The compiler has surpassed the maximum number of errors which can occur in a single compilation. The only solution is to address some of the errors and recompile the project.
E2034: Too many actual parameters (see page 490)	This error message occurs when a procedure or function call gives more parameters than the procedure or function declaration specifies. Additionally, this error message occurs when an OLE automation call has too many (more than 255), or too many named parameters.
E2436: Type declarations not allowed in anonymous record or local record type (see page 491)	Record types that are declared in local scopes or declared in-place in variable declarations can only contain field declarations. For advanced features in record types (such as methods, properties, and nested types), the record type must be an explicitly declared global type.
E2005: '%s' is not a type identifier (see page 491)	This error message occurs when the compiler expected the name of a type, but the name it found did not stand for a type.
x2243: Expression needs no Initialize/Finalize (see page 491)	You have attempted to use the standard procedure Finalize on a type that requires no finalization.
E2100: Data type too large: exceeds 2 GB (see page 492)	You have specified a data type which is too large for the compiler to represent. The compiler will generate this error for datatypes which are greater or equal to 2 GB in size. You must decrease the size of the description of the type.

E2101: Size of data type is zero (↗ see page 492)	Record types must contain at least one instance data field. Zero-size records are not allowed in .NET.
W1016: Typed constant '%s' passed as var parameter (↗ see page 492)	This error message is reserved.
W1055: Published caused RTTI ($M+) to be added to type '%s' (↗ see page 492)	You added a 'PUBLISHED' section to a class that was not compiled while the {$M+}/{$TYPEINFO ON} switch was in effect, or without deriving from a class compiled with the {$M+}/{$TYPEINFO ON} switch in effect. The TypeInfo standard procedure requires a type identifier as its parameter. In the code above, 'NotType' does not represent a type identifier. To avoid this error, ensure that you compile while the {$M+}/{$TYPEINFO ON} switch is on, or derive from a class that was compiled with {$M+}/{$TYPEINFO ON} switch on.
E2133: TYPEINFO standard function expects a type identifier (↗ see page 493)	You have attempted to obtain type information for an identifier which does not represent a type.
E2147: Property '%s' does not exist in base class (↗ see page 493)	The compiler believes you are attempting to hoist a property to a different visibility level in a derived class, but the specified property does not exist in the base class.
E2452: Unicode characters not allowed in published symbols (↗ see page 494)	The VCL Run-Time Type Information (RTTI) subsystem and the streaming of DFM files require that published symbols are non-Unicode (ANSI) characters. Consider whether this symbol needs to be published, and if so, use ANSI characters instead of Unicode.
W1041: Error converting Unicode char to locale charset. String truncated. Is your LANG environment variable set correctly? (↗ see page 494)	This message occurs when you are trying to convert strings in Unicode to your local character set and the string contains characters that are not valid for the current locale. For example, this may occur when converting WideString to AnsiString or if attempting to display Japanese characters in an English locale.
W1006: Unit '%s' is deprecated (↗ see page 494)	The unit is deprecated, but continues to be available to support backward compatibility. The unit is tagged (using the **deprecated** hint directive) as no longer current and is maintained for compatibility only. You should consider updating your source code to use another unit, if possible. The **$WARN** UNIT_DEPRECATED ON/OFF compiler directive turns on or off all warnings about the **deprecated** directive in units where the **deprecated** directive is specified.
W1007: Unit '%s' is experimental (↗ see page 494)	An "experimental" directive has been used on an identifier. "Experimental" indicates the presence of a class or unit which is incomplete or not fully tested.
F2048: Bad unit format: '%s' (↗ see page 495)	This error occurs when a compiled unit file (.dcu file) has a bad format. Most likely, the file has been corrupted. Recompile the file if you have the source. If the problem persists, you may have to reinstall Delphi.
W1052: Can't find System.Runtime.CompilerServices.RunClassConstructor. Unit initialization order will not follow uses clause order (↗ see page 495)	This warning indicates that the initialization order defined by the Delphi language, that specified by the order of units in the uses clause, is not guaranteed. The RunClassConstructor function is used to execute the initialization sections of units used by the current unit in the order specified by the current unit's **uses** clauses. This warning will be issued if the compiler cannot find this function in the .NET Framework you are linking against. For example, it will occur when linking against the .NET Compact Framework, which does not implement RunClassConstructor.
W1004: Unit '%s' is specific to a library (↗ see page 495)	The whole unit is tagged (using the **library** hint directive) as one that may not be available in all libraries. If you are likely to use different libraries, it may cause a problem. The **$WARN** UNIT_LIBRARY ON/OFF compiler directive turns on or off all warnings in units where the **library** directive is specified.
E1038: Unit identifier '%s' does not match file name (↗ see page 495)	The unit name in the top unit is case sensitive and must match the name with respect to upper- and lowercase letters exactly. The unit name is case sensitive only in the unit declaration.
W1005: Unit '%s' is specific to a platform (↗ see page 495)	The whole unit is tagged (using the **platform** hint directive) as one that contains material that may not be available on all platforms. If you are writing cross-platform applications, it may cause a problem. For example, a unit that uses objects defined in OleAuto might be tagged using the PLATFORM directive The **$WARN** UNIT_PLATFORM ON/OFF compiler directive turns on or off all warnings about the **platform** directive in units where the **platform** directive is specified.
E2070: Unknown directive: '%s' (↗ see page 495)	This error message appears when the compiler encounters an unknown directive in a procedure or function declaration. The directive is probably misspelled, or a semicolon is missing.
E2328: Linker error while emitting metadata (↗ see page 496)	No further information is available for this error or warning.
E2400: Unknown Resource Format '%s' (↗ see page 496)	No further information is available for this error or warning.
E2216: Can't handle section '%s' in object file '%s' (↗ see page 496)	You are trying to link object modules into your program with the $L compiler directive. However, the object file is too complex for the compiler to handle. For example, you may be trying to link in a C++ object file. This is not supported.

E2405: Unknown element type found importing signature of %s.%s (↗ see page 496)	No further information is available for this error or warning.
E2417: Field offset cannot be determined for variant record because previous field type is unknown size record type (↗ see page 496)	Private types in an assembly are not imported and are marked as having an unreliable size. If a record is declared as having at least one private field or it has one field whose type size is unreliable then this error will occur.
E2166: Unnamed arguments must precede named arguments in OLE Automation call (↗ see page 496)	You have attempted to follow named OLE Automation arguments with unnamed arguments.
E2289: Unresolved custom attribute: %s (↗ see page 497)	A custom attribute declaration was not followed by a symbol declaration such as a type, variable, method, or parameter declaration.
W1048: Unsafe typecast of '%s' to '%s' (↗ see page 497)	You have used a data type or operation for which static code analysis cannot prove that it does not overwrite memory. In a secured execution environment such as .NET, such code is assumed to be unsafe and a potential security risk.
W1047: Unsafe code '%s' (↗ see page 497)	You have used a data type or operation for which static code analysis cannot prove that it does not overwrite memory. In a secured execution environment such as .NET, such code is assumed to be unsafe and a potential security risk.
E2406: EXPORTS section allowed only if compiling with {$UNSAFECODE ON} (↗ see page 497)	No further information is available for this error or warning.
W1046: Unsafe type '%s%s%s' (↗ see page 497)	You have used a data type or operation for which static code analysis cannot prove that it does not overwrite memory. In a secured execution environment such as .NET, such code is assumed to be unsafe and a potential security risk.
E2396: Unsafe code only allowed in unsafe procedure (↗ see page 497)	No further information is available for this error or warning.
E2395: Unsafe procedure only allowed if compiling with {$UNSAFECODE ON} (↗ see page 498)	No further information is available for this error or warning.
E2397: Unsafe pointer only allowed if compiling with {$UNSAFECODE ON} (↗ see page 498)	No further information is available for this error or warning.
E2410: Unsafe pointer variables, parameters or consts only allowed in unsafe procedure (↗ see page 498)	No further information is available for this error or warning.
x1025: Unsupported language feature: '%s' (↗ see page 498)	You are attempting to translate a Delphi unit to a C++ header file which contains unsupported language features. You must remove the offending construct from the interface section before the unit can be translated.
E2057: Unexpected end of file in comment started on line %ld (↗ see page 498)	This error occurs when you open a comment, but do not close it. Note that a comment started with '{' must be closed with '}', and a comment started with '(*' must be closed with '*)'.
E2280: Unterminated conditional directive (↗ see page 498)	For every {$IFxxx}, the corresponding {$ENDIF} or {$IFEND} must be found within the same source file. This message indicates that you do not have an equal number of ending directives. This error message is reported at the source line of the last $IF/$IFDEF/etc. with no matching $ENDIF/$IFEND. This gives you a good place to start looking for the source of the problem.
E2052: Unterminated string (↗ see page 498)	The compiler did not find a closing apostrophe at the end of a character string. Note that character strings cannot be continued onto the next line - however, you can use the '+' operator to concatenate two character strings on separate lines.
H2164: Variable '%s' is declared but never used in '%s' (↗ see page 499)	You have declared a variable in a procedure, but you never actually use it. -H
W1036: Variable '%s' might not have been initialized (↗ see page 499)	This warning is given if a variable has not been assigned a value on every code path leading to a point where it is used.
E2157: Element 0 inaccessible - use 'Length' or 'SetLength' (↗ see page 501)	The Delphi String type does not store the length of the string in element 0. The old method of changing, or getting, the length of a string by accessing element 0 does not work with long strings.
E2255: New not supported for dynamic arrays - use SetLength (↗ see page 502)	The program has attempted to use the standard procedure NEW on a dynamic array. The proper method for allocating dynamic arrays is to use the standard procedure SetLength.
E2212: Package unit '%s' cannot appear in contains or uses clauses (↗ see page 502)	The unit named in the error is a package unit and as such cannot be included in your project. A possible cause of this error is that somehow a Delphi unit and a package unit have been given the same name. The compiler is finding the package unit on its search path before it can locate a same-named Delphi file. Packages cannot be included in a project by inclusion of the package unit in the uses clause.
F2063: Could not compile used unit '%s' (↗ see page 502)	This fatal error is given when a unit used by another could not be compiled. In this case, the compiler gives up compilation of the dependent unit because it is likely very many errors will be encountered as a consequence.
E2090: User break - compilation aborted (↗ see page 502)	This message is currently unused.
E2165: Compile terminated by user (↗ see page 502)	You pressed Ctrl-Break during a compile.
E2142: Inaccessible value (↗ see page 502)	You have tried to view a value that is not accessible from within the integrated debugger. Certain types of values, such as a 0 length Variant-type string, cannot be viewed within the debugger.

H2077: Value assigned to '%s' never used (↗ see page 503)	The compiler gives this hint message if the value assigned to a variable is not used. If optimization is enabled, the assignment is eliminated. This can happen because either the variable is not used anymore, or because it is reassigned before it is used.
E2088: Variable name expected (↗ see page 504)	This error message is issued if you try to declare an absolute variable, but the absolute directive is not followed by an integer constant or a variable name.
E2171: Variable '%s' inaccessible here due to optimization (↗ see page 504)	The evaluator or watch statement is attempting to retrieve the value of <name>, but the compiler was able to determine that the variables actual lifetime ended prior to this inspection point. This error will often occur if the compiler determines a local variable is assigned a value that is not used beyond a specific point in the program's control flow.
E2033: Types of actual and formal var parameters must be identical (↗ see page 505)	For a variable parameter, the actual argument must be of the exact type of the formal parameter.
E2277: Only external cdecl functions may use varargs (↗ see page 505)	This message indicates that you are trying to implement a varargs routine. You cannot implement varargs routines, you can only call external varargs.
F2051: Unit %s was compiled with a different version of %s.%s (↗ see page 505)	This fatal error occurs when the declaration of symbol declared in the interface part of a unit has changed, and the compiler cannot recompile a unit that relies on this declaration because the source is not available to it. There are several possible solutions - recompile Unit1 (assuming you have the source code available), use an older version of Unit2 or change Unit2, or get a new version of Unit1 from whoever has the source code. This error can also occur when a unit in your project has the same name as a standard Delphi unit. For example, this may... more (↗ see page 505)
E2379: Virtual methods not allowed in record types (↗ see page 506)	No further information is available for this error or warning.
E2423: Void type not usable in this context (↗ see page 506)	The System type Void is not allowed to be used in some contexts. As an example, the following code demostrates the contexts where type Void may not be used.
E2221: $WEAKPACKAGEUNIT '%s' cannot have initialization or finalization code (↗ see page 506)	A unit which has been flagged with the $weakpackageunit directive cannot contain initialization or finalization code, nor can it contain global data. The reason for this is that multiple copies of the same weakly packaged units can appear in an application, and then referring to the data for that unit becomes and ambiguous proposition. This ambiguity is furthered when dynamically loaded packages are used in your applications.
E2203: $WEAKPACKAGEUNIT '%s' contains global data (↗ see page 507)	A unit which was marked with $WEAKPACKAGEUNIT is being placed into a package, but it contains global data. It is not legal for such a unit to contain global data or initialization or finalization code. The only solutions to this problem are to remove the $WEAKPACKAGEUNIT mark, or remove the global data from the unit before it is put into the package.
W1050: WideChar reduced to byte char in set expressions (↗ see page 507)	"Set of char" in Win32 defines a set over the entire range of the Char type. Since Char is a byte-sized type in Win32, this defines a set of maximum size containing 256 elements. In .NET, Char is a word-sized type, and this range (0..65535) exceeds the capacity of the set type. To accomodate existing code that uses this "Set of Char" syntax, the compiler will treat the expression as "set of AnsiChar". The warning message reminds you that the set can only store the boolean state of 256 distinct elements, not the full range of the Char type.
E2152: Wrong or corrupted version of RLINK32.DLL (↗ see page 507)	The internal consistency check performed on the RLINK32.DLL file has failed. Contact CodeGear if you encounter this error.
E2015: Operator not applicable to this operand type (↗ see page 507)	This error message is given whenever an operator cannot be applied to the operands it was given - for instance if a boolean operator is applied to a pointer.
W1206: XML comment on '%s' has cref attribute '%s' that could not be resolved (↗ see page 508)	This warning message occurs when the XML has a cref attribute that cannot be resolved. This is a warning in XML documentation processing. The XML is well formed, but the comment's meaning is questionable. XML cref references follow the .NET style. See http://msdn2.microsoft.com/en-us/library/acd0tfbe.aspx for more details. A documentation warning does not prevent building.
W1205: XML comment on '%s' has badly formed XML--'The character '%c' was expected.' (↗ see page 508)	This warning message occurs when the expected character was not found in the XML. This is an error in XML documentation processing. The XML is not well formed. This is a warning because a documentation error does not prevent building.
W1204: XML comment on '%s' has badly formed XML--'A name contained an invalid character.' (↗ see page 508)	This warning message occurs when a name in XML contains an invalid character. This is an error in XML documentation processing. The XML is not well formed. This is a warning because a documentation error does not prevent building.
W1203: XML comment on '%s' has badly formed XML--'A name was started with an invalid character.' (↗ see page 508)	This warning message occurs when an XML name was started with an invalid character. This is an error in XML documentation processing. The XML is not well formed. This is a warning because a documentation error does not prevent building.

W1208: Parameter '%s' has no matching param tag in the XML comment for '%s' (but other parameters do) (↗ see page 508)	This warning message occurs when an XML Parameter has no matching param tag in the XML comment but other parameters do. This is a warning in XML documentation processing. There is at least one tag, but some parameters in the method don't have a tag. A documentation warning does not prevent building.
W1207: XML comment on '%s' has a param tag for '%s', but there is no parameter by that name (↗ see page 508)	This warning message occurs when the XML contains a parameter tag for a nonexistent parameter. This is a warning in XML documentation processing. The XML is well formed, however, a tag was created for a parameter that doesn't exist in a method. A documentation warning does not prevent building.
W1202: XML comment on '%s' has badly formed XML--'Reference to undefined entity '%s'' (↗ see page 509)	This warning message occurs when XML references an undefined entity. This is an error in XML documentation processing. The XML is not well formed. This is a warning because a documentation error does not prevent building.
W1201: XML comment on '%s' has badly formed XML--'Whitespace is not allowed at this location.' (↗ see page 509)	This warning message occurs when the compiler encounters white space in a location in which white space is not allowed. This is an error in XML documentation processing. The XML is not well formed. This is a warning because a documentation error does not prevent building.
W1013: Constant 0 converted to NIL (↗ see page 509)	The Delphi compiler now allows the constant 0 to be used in pointer expressions in place of NIL. This change was made to allow older code to still compile with changes which were made in the low-level RTL.

3.1.2.1.1 DisposeCount cannot be declared in classes with destructors

No further Help is available for this message or warning.

3.1.2.1.2 E2190: Thread local variables cannot be ABSOLUTE

A thread local variable cannot refer to another variable, nor can it reference an absolute memory address.

```
program Produce;

  threadvar
     secretNum : integer absolute $151;

begin
end.
```

The absolute directive is not allowed in a threadvar declaration section.

```
program Solve;

  threadvar
    secretNum : integer;

  var
    sNum : integer absolute $151;

begin
end.
```

There are two easy ways to solve a problem of this nature. The first is to remove the absolute directive from the threadvar section. The second would be to move the absolute variable to a normal var declaration section.

3.1.2.1.3 E2249: Cannot use absolute variables when compiling to byte code

The use of absolute variables is prohibited when compiling to byte code.

3.1.2.1.4 E2373: Call to abstract method %s.%s

No further information is available for this error or warning.

3.1.2.1.5 E2371: ABSTRACT and FINAL cannot be used together

A class cannot be both final and abstract.

Final is a restrictive modifier used to prevent extension of a class (or prevent overrides on methods), while the abstract modifier signals the intention to use a class as a base class.

3.1.2.1.6 E2136: No definition for abstract method '%s' allowed

You have declared <name> to be abstract, but the compiler has found a definition for the method in the source file. It is illegal to provide a definition for an abstract declaration.

```
program Produce;

  type
    Base = class
      procedure Foundation; virtual; abstract;
    end;

    procedure Base.Foundation;
    begin
    end;

begin
end.
```

Abstract methods cannot be defined. An error will appear at the point of Base.Foundation when you compile this program.

```
program Solve;

  type
    Base = class
      procedure Foundation; virtual; abstract;
    end;

    Derived = class (Base)
      procedure Foundation; override;
    end;

    procedure Derived.Foundation;
    begin
    end;

begin
end.
```

Two steps are required to solve this error. First, you must remove the definition of the abstract procedure which is declared in the base class. Second, you must extend the base class, declare the abstract procedure as an 'override' in the extension, and then provide a definition for the newly declared procedure.

3.1.2.1.7 E2167: Abstract methods must be virtual or dynamic

When declaring an abstract method in a base class, it must either be of regular virtual or dynamic virtual type.

```
program Produce;

  type
    Base = class
      procedure DaliVision; abstract;
      procedure TellyVision; abstract;
    end;
```

```
begin
end.
```

The declaration above is in error because abstract methods must either be virtual or dynamic.

```
program Solve;

  type
    Base = class
      procedure DaliVision; virtual; abstract;
      procedure TellyVision; dynamic; abstract;
    end;

begin
end.
```

It is possible to remove this error by either specifying 'virtual' or 'dynamic', whichever is most appropriate for your application.

3.1.2.1.8 E2383: ABSTRACT and SEALED cannot be used together

A class cannot be both sealed and abstract.

The sealed modifier is used to prevent inheritance of a class, while the abstract modifier signals the intention to use a class as a base class.

3.1.2.1.9 E2247: Cannot take the address when compiling to byte code

The address-of operator, @, cannot be used when compiling to byte code.

3.1.2.1.10 E2251: Ambiguous overloaded call to '%s'

Based on the current overload list for the specified function, and the programmed invocation, the compiler is unable to determine which version of the procedure should be invoked.

```
program Produce;

procedure f0(a : integer); overload;
begin
end;

procedure f0(a : integer; b : char = 'A'); overload;
begin
end;

begin
  f0(1);
end.
```

In this example, the default parameter that exists in one of the versions of f0 makes it impossible for the compiler to determine which procedure should actually be called.

```
program Solve;

procedure f0(a : integer); overload;
begin
end;

procedure f0(a : integer; b : char); overload;
begin
end;
```

```
begin
  f0(1);
end.
```

The approach taken in this example was to remove the default parameter value. The result here is that the procedure taking only one integer parameter will be called. It should be noted that this approach is the only way that the single-parameter function can be called.

3.1.2.1.11 E2099: Overflow in conversion or arithmetic operation

The compiler has detected an overflow in an arithmetic expression: the result of the expression is too large to be represented in 32 bits.

Check your computations to ensure that the value can be represented by the computer hardware.

3.1.2.1.12 E2307: NEW standard function expects a dynamic array type identifier

No further information is available for this error or warning.

3.1.2.1.13 E2308: Need to specify at least one dimension for NEW of dynamic array

No further information is available for this error or warning.

3.1.2.1.14 E2246: Need to specify at least one dimension for SetLength of dynamic array

The standard procedure SetLength has been called to alter the length of a dynamic array, but no array dimensions have been specified.

```
program Produce;

  var
    arr : array of integer;

begin
  SetLength(arr);
end.
```

The SetLength in the above example causes an error since no array dimensions have been specified.

```
program solve;

  var
    arr : array of integer;

begin
  SetLength(arr, 151);
end.
```

To remove this error from your program, specify the number of elements you want the array to contain.

3.1.2.1.15 E2081: Assignment to FOR-Loop variable '%s'

It is illegal to assign a value to the for loop control variable inside the for loop.

If the purpose is to leave the loop prematurely, use a break or goto statement.

```
program Produce;

var
  I: Integer;
  A: array [0..99] of Integer;
begin
  for I := 0 to 99 do begin
    if A[I] = 42 then
      I := 99;
  end;
end.
```

In this case, the programmer thought that assigning 99 to I would cause the program to exit the loop.

```
program Solve;

var
  I: Integer;
  A: array [0..99] of Integer;
begin
  for I := 0 to 99 do begin
    if A[I] = 42 then
      Break;
  end;
end.
```

Using a break statement is a cleaner way to exit out of a for loop.

3.1.2.1.16 W1017: Assignment to typed constant '%s'

This warning message is currently unused.

3.1.2.1.17 E2117: 486/487 instructions not enabled

You should not receive this error as 486 instructions are always enabled.

3.1.2.1.18 E2116: Invalid combination of opcode and operands

You have specified an inline assembler statement which is not correct.

```
program Produce;

  procedure AssemblerExample;
  asm
    mov al, $0f0 * 16
  end;

begin
end.
```

The inline assembler is not capable of storing the result of $f0 * 16 into the 'al' register—it simply won't fit.

```
program Solve;
  procedure AssemblerExample;
  asm
```

```
    mov al, $0f * 16
  end;

begin
end.
```

Make sure that the type of both operands are compatible.

3.1.2.1.19 E2109: Constant expected

The inline assembler was expecting to find a constant but did not find one.

```
program Produce;

  procedure Assembly(x : Integer);
  asm
    mov   ax, x MOD 10
  end;

begin
end.
```

The inline assembler is not capable of performing a MOD operation on a Delphi variable, thus the above code will cause an error.

Many of the inline assembler expressions require constants to assemble correctly. Change the offending statement to have a assemble-time constant.

3.1.2.1.20 E2118: Division by zero

The inline assembler has encountered an expression which results in a division by zero.

```
program Produce;

  procedure AssemblerExample;
  asm
    dw  1000 / 0
  end;

begin
end.
```

If you are using program constants instead of constant literals, this error might not be quite so obvious.

```
program Solve;

  procedure AssemblerExample;
  asm
    dw  1000 / 10
  end;

begin
end.
```

The solution, as when programming in high-level languages, is to make sure that you don't divide by zero.

3.1.2.1.21 E2119: Structure field identifier expected

The inline assembler recognized an identifier on the right side of a '.', but it was not a field of the record found on the left side of the '.'. One common, yet difficult to realize, error of this sort is to use a record with a field called 'ch' - the inline assembler will always interpret 'ch' to be a register name.

```
program Produce;

  type
    Data = record
      x : Integer;
    end;

  procedure AssemblerExample(d : Data; y : Char);
  asm
    mov  eax, d.y
  end;

begin
end.
```

In this example, the inline assembler has recognized that 'y' is a valid identifier, but it has not found 'y' to be a member of the type of 'd'.

```
program Solve;

  type
    Data = record
      x : Integer;
    end;

  procedure AssemblerExample(d : Data; y : Char);
  asm
    mov  eax, d.x
  end;

begin
end.
```

By specifying the proper variable name, the error will go away.

3.1.2.1.22 E2108: Memory reference expected

The inline assembler has expected to find a memory reference expression but did not find one.

Ensure that the offending statement is indeed a memory reference.

3.1.2.1.23 E2115: Error in numeric constant

The inline assembler has found an error in the numeric constant you entered.

```
program Produce;

  procedure AssemblerExample;
  asm
    mov al, $z0f0
  end;

begin
end.
```

In the example above, the inline assembler was expecting to parse a hexadecimal constant, but it found an erroneous character.

```
program Solve;

  procedure AssemblerExample;
  asm
    mov al, $f0
  end;
```

```
begin
end.
```

Make sure that the numeric constants you enter conform to the type that the inline assembler is expecting to parse.

3.1.2.1.24 E2107: Operand size mismatch

The size required by the instruction operand does not match the size given.

```
program Produce;

  var
    v : Integer;

  procedure Assembly;
  asm
    db offset v
  end;

begin
end.
```

In the sample above, the compiler will complain because the 'offset' operator produces a 'dword', but the operator is expecting a 'byte'.

```
program Solve;

  var
    v : Integer;

  procedure Assembly;
  asm
    dd offset v
  end;

begin
end.
```

The solution, for this example, is to change the operator to receive a 'dword'. In the general case, you will need to closely examine your code and ensure that the operator and operand sizes match.

3.1.2.1.25 E2113: Numeric overflow

The inline assembler has detected a numeric overflow in one of your expressions.

```
program Produce;

  procedure AssemblerExample;
  asm
    mov eax, $0ffffffffffffffffffffff
  end;

begin
end.
```

Specifying a number which requires more than 32 bits to represent will elicit this error.

```
program Solve;

  procedure AssemblerExample;
  asm
    mov al, $0ff
  end;
```

```
begin
end.
```

Make sure that your numbers all fit in 32 bits.

3.1.2.1.26 E2112: Invalid register combination

You have specified an illegal combination of registers in a inline assembler statement. Please refer to an assembly language guide for more information on addressing modes allowed on the Intel 80x86 family.

```
program Produce;

  procedure AssemblerExample;
  asm
    mov eax, [ecx + esp * 4]
  end;

begin
end.
```

The right operand specified in this mov instruction is illegal.

```
program Solve;

  procedure AssemblerExample;
  asm
    mov eax, [ecx + ebx * 4]
  end;

begin
end.
```

The addressing mode specified by the right operand of this mov instruction is allowed.

3.1.2.1.27 E2111: Cannot add or subtract relocatable symbols

The inline assembler is not able to add or subtract memory address which may be changed by the linker.

```
program Produce;

  var
    a : array [1..10] of Integer;
    endOfA : Integer;

  procedure Relocatable;
  begin
  end;

  procedure Assembly;
  asm
    mov eax, a + endOfA
  end;

begin
end.
```

Global variables fall into the class of items which produce relocatable addresses, and the inline assembler is unable to add or subtract these.

Make sure you don't try to add or subtract relocatable addresses from within your inline assembler statements.

3.1.2.1.28 E2106: Inline assembler stack overflow

Your inline assembler code has exceeded the capacity of the inline assembler.

Contact CodeGear if you encounter this error.

3.1.2.1.29 E2114: String constant too long

The inline assembler has not found the end of the string that you specified. The most likely cause is a misplaced closing quote.

```
program Produce;

  procedure AssemblerExample;
  asm
    db 'Hello world.  I am an inline assembler statement
  end;

begin
end.
```

The inline assembler is unable to find the end of the string, before the end of the line, so it reports that the string is too long.

```
program Solve;

  procedure AssemblerExample;
  asm
    db 'Hello world.  I am an inline assembler statement'
  end;

begin
end.
```

Adding the closing quote will vanquish this error.

3.1.2.1.30 E2105: Inline assembler syntax error

You have entered an expression which the inline assembler is unable to interpret as a valid assembly instruction.

```
program Produce;

  procedure Assembly;
  asm
    adx  eax, 151
  end;

begin
end.
    program Solve;

  procedure Assembly;
  asm
    add  eax, 151
  end;

begin
end.
```

Examine the offending inline assembly statement and ensure that it conforms to the proper syntax.

3.1.2.1.31 E2110: Type expected

Contact CodeGear if you receive this error.

3.1.2.1.32 E2448: An attribute argument must be a constant expression, typeof expression or array constructor

The Common Language Runtime specifies that an attribute argument must be a constant expression, a typeof expression or an array creation expression. Attribute arguments cannot be global variables, for example. Attribute instances are constructed at compile-time and incorporated into the assembly metadata, so no run-time information can be used to construct them.

3.1.2.1.33 E2045: Bad object file format: '%s'

This error occurs if an object file loaded with a $L or $LINK directive is not of the correct format. Several restrictions must be met:

- Check the naming restrictions on segment names in the help file
- Not more than 10 segments
- Not more than 255 external symbols
- Not more than 50 local names in LNAMES records
- LEDATA and LIDATA records must be in offset order
- No THREAD subrecords are supported in FIXU32 records
- Only 32-bit offsets can be fixed up
- Only segment and self relative fixups
- Target of a fixup must be a segment, a group or an EXTDEF
- Object must be 32-bit object file
- Various internal consistency condition that should only fail if the object file is corrupted.

3.1.2.1.34 x1028: Bad global symbol definition: '%s' in object file '%s'

This warning is given when an object file linked in with a $L or $LINK directive contains a definition for a symbol that was not declared in Delphi as an external procedure, but as something else (e.g. a variable).

The definition in the object will be ignored in this case.

3.1.2.1.35 E2160: Type not allowed in OLE Automation call

If a data type cannot be converted by the compiler into a Variant, then it is not allowed in an OLE automation call.

```
program Produce;

  type
    Base = class
      x : Integer;
    end;

  var
    B : Base;
    V : Variant;
```

```
begin
  V.Dispatch(B);
end.
```

A class cannot be converted into a Variant type, so it is not allowed in an OLE call.

```
program Solve;


  type
    Base = class
      x : Integer;
    end;

  var
    B : Base;
    V : Variant;

begin
  V.Dispatch(B.i);
end.
```

The only solution to this problem is to manually convert these data types to Variants or to only use data types that can automatically be converted into a Variant.

3.1.2.1.36 E2188: Published property '%s' cannot be of type %s

Published properties must be an ordinal type, Single, Double, Extended, Comp, a string type, a set type which fits in 32 bits, or a method pointer type. When any other property type is encountered in a published section, the compiler will remove the published attribute -$M+

```
(*$TYPEINFO ON*)
program Produce;

  type
    TitleArr = array [0..24] of char;
    NamePlate = class
    private
      titleStr : TitleArr;
    published
      property Title : TitleArr read titleStr write titleStr;
    end;

begin
end.
```

An error is induced because an array is not one of the data types which can be published.

```
(*$TYPEINFO ON*)
program Solve;

  type
    TitleArr = integer;
    NamePlate = class
      titleStr : TitleArr;
    published
      property Title : TitleArr read titleStr write titleStr;
    end;

begin
end.
```

Moving the property declaration out of the published section will avoid this error. Another alternative, as in this example, is to change the type of the property to be something that can actually be published.

3.1.2.1.37 E2055: Illegal type in Read/Readln statement

This error occurs when you try to read a variable in a Read or Readln that is not of a legal type.

Check the type of the variable and make sure you are not missing a dereferencing, indexing or field selection operator.

```
program Produce;
type
  TColor = (red,green,blue);
var
  Color : TColor;
begin
  Readln(Color);     (*<-- Error message here*)
end.
```

We cannot read variables of enumerated types directly.

```
program Solve;
type
  TColor = (red,green,blue);
var
  Color : TColor;
  InputString: string;
const
  ColorString : array [TColor] of string = ('red', 'green', 'blue');
begin
  Readln(InputString);
  Color := red;
  while (color < blue) and (ColorString[color] <> InputString) do
    Inc(color);
end.
```

The solution is to read a string, and look up that string in an auxiliary table. In the example above, we didn't bother to do error checking - any string will be treated as 'blue'. In practice, we would probably output an error message and ask the user to try again.

3.1.2.1.38 E2053: Syntax error in real number

This error message occurs if the compiler finds the beginning of a scale factor (an 'E' or 'e' character) in a number, but no digits follow it.

```
program Produce;
const
  SpeedOfLight = 3.0E 8;    (*<-- Error message here*)
begin
end.
```

In the example, we put a space after '3.0E' - now for the compiler the number ends here, and it is incomplete.

```
program Solve;
const
  SpeedOfLight = 3.0E+8;
begin
end.
```

We could have just deleted the blank, but we put in a '+' sign because it looks nicer.

3.1.2.1.39 E2104: Bad relocation encountered in object file '%s'

You are trying to link object modules into your program with the $L compiler directive. However, the object file is too complex for the compiler to handle. For example, you may be trying to link in a C++ object file. This is not supported.

3.1.2.1.40 E2158: %s unit out of date or corrupted: missing '%s'

The compiler is looking for a special function which resides in System.dcu but could not find it. Your System unit is either corrupted or obsolete.

Make sure there are no conflicts in your library search path which can point to another System.dcu. Try reinstalling System.dcu. If neither of these solutions work, contact CodeGear Developer Support.

3.1.2.1.41 E2159: %s unit out of date or corrupted: missing '%s.%s'

The compiler failed to find a special function in System, indicating that the unit found in your search paths is either corrupted or obsolete.

3.1.2.1.42 E2150: Bad argument type in variable type array constructor

You are attempting to construct an array using a type which is not allowed in variable arrays.

```
program Produce;

  type
    Fruit = (apple, orange, pear);
    Data = record
      x : Integer;
      ch : Char;
    end;

  var
    f : Fruit;
    d : Data;

  procedure Examiner(v : array of TVarRec);
  begin
  end;

begin
  Examiner([d]);
  Examiner([f]);
end.
```

Both calls to Examiner will fail because enumerations and records are not supported in array constructors.

```
program Solve;

  var
    i : Integer;
    r : Real;
    v : Variant;

  procedure Examiner(v : array of TVarRec);
  begin
  end;

begin
  i := 0; r := 0; v := 0;
  Examiner([i, r, v]);
end.
```

Many data types, like those in the example above, are allowed in array constructors.

3.1.2.1.43 E2281: Type not allowed in Variant Dispatch call

This message indicates that you are trying to make a method call and are passing a type that the compiler does not know how to marshall. Variants can hold interfaces, but the interfaces can marshall only certain types.

On Windows, Delphi supports COM and SOAP interfaces and can call types that these interfaces can marshall.

3.1.2.1.44 E2054: Illegal type in Write/Writeln statement

This error occurs when you try to output a type in a Write or Writeln statement that is not legal.

```
program Produce;
type
  TColor = (red,green,blue);
var
  Color : TColor;
begin
  Writeln(Color);
end.
```

It would have been convenient to use a writeln statement to output Color, wouldn't it?

```
program Solve;
type
  TColor = (red,green,blue);
var
  Color : TColor;
const
  ColorString : array [TColor] of string = ('red', 'green', 'blue');
begin
  Writeln(ColorString[Color]);
end.
```

Unfortunately, that is not legal, and we have to do it with an auxiliary table.

3.1.2.1.45 E2297: Procedure definition must be ILCODE calling convention

.NET managed code can only use the ILCODE calling convention.

3.1.2.1.46 E2050: Statements not allowed in interface part

The interface part of a unit can only contain declarations, not statements.

Move the bodies of procedures to the implementation part.

```
unit Produce;

interface

procedure MyProc;
begin                (*<-- Error message here*)
end;

implementation

begin
end.
```

We got carried away and gave MyProc a body right in the interface section.

```
unit Solve;
```

```
interface

procedure MyProc;

implementation

procedure MyProc;
begin
end;

begin
end.
```

We need move the body to the implementation section - then it's fine.

3.1.2.1.47 x1012: Constant expression violates subrange bounds

This error message occurs when the compiler can determine that a constant is outside the legal range. This can occur for instance if you assign a constant to a variable of subrange type.

```
program Produce;
var
  Digit: 1..9;
begin
  Digit := 0;  (*Get message: Constant expression violates subrange bounds*)
end.
    program Solve;
var
  Digit: 0..9;
begin
  Digit := 0;
end.
```

3.1.2.1.48 E2097: BREAK or CONTINUE outside of loop

The compiler has found a BREAK or CONTINUE statement which is not contained inside a WHILE or REPEAT loop. These two constructs are only legal in loops.

```
program Produce;

  procedure Error;
    var i : Integer;
  begin
    i := 0;
    while i < 100 do
      INC(i);
      if odd(i) then begin
        INC(i);
  continue;
      end;
  end;

begin
end.
```

The example above shows how a continue statement could seem to be included in the body of a looping construct but, due to the compound-statement nature of The Delphi language, it really is not.

```
program Solve;

  procedure Error;
    var i : Integer;
  begin
```

```
      i := 0;
      while i < 100 do begin
        INC(i);
        if odd(i) then begin
          INC(i);
    continue;
        end;
      end;
    end;

  begin
  end.
```

Often times it is a simple matter to create compound statement out of the looping construct to ensure that your CONTINUE or BREAK statements are included.

3.1.2.1.49 E2309: Attribute - Known attribute named argument cannot be an array

No further information is available for this error or warning.

3.1.2.1.50 E2310: Attribute - A custom marshaler requires the custom marshaler type

No further information is available for this error or warning.

3.1.2.1.51 E2327: Linker error while emitting attribute '%s' for '%s'

No further information is available for this error or warning.

3.1.2.1.52 E2311: Attribute - MarshalAs fixed string requires a size

No further information is available for this error or warning.

3.1.2.1.53 E2312: Attribute - Invalid argument to a known attribute

No further information is available for this error or warning.

3.1.2.1.54 E2313: Attribute - Known attribute cannot specify properties

No further information is available for this error or warning.

3.1.2.1.55 E2314: Attribute - The MarshalAs attribute has fields set that are not valid for the specified unmanaged type

No further information is available for this error or warning.

3.1.2.1.56 E2315: Attribute - Known custom attribute on invalid target

No further information is available for this error or warning.

3.1.2.1.57 E2316: Attribute - The format of the GUID was invalid

No further information is available for this error or warning.

3.1.2.1.58 E2317: Attribute - Known custom attribute had invalid value

No further information is available for this error or warning.

3.1.2.1.59 E2318: Attribute - The MarshalAs constant size cannot be negative

No further information is available for this error or warning.

3.1.2.1.60 E2319: Attribute - The MarshalAs parameter index cannot be negative

No further information is available for this error or warning.

3.1.2.1.61 E2320: Attribute - The specified unmanaged type is only valid on fields

No further information is available for this error or warning.

3.1.2.1.62 E2321: Attribute - Known custom attribute has repeated named argument

No further information is available for this error or warning.

3.1.2.1.63 E2322: Attribute - Unexpected type in known attribute

No further information is available for this error or warning.

3.1.2.1.64 E2323: Attribute - Unrecognized argument to a known custom attribute

No further information is available for this error or warning.

3.1.2.1.65 E2324: Attribute - Known attribute named argument doesn't support variant

No further information is available for this error or warning.

3.1.2.1.66 E2222: $WEAKPACKAGEUNIT & $DENYPACKAGEUNIT both specified

It is not legal to specify both $WEAKPACKAGEUNIT and $DENYPACKAGEUNIT. Correct the source code and recompile.

3.1.2.1.67 E2276: Identifier '%s' cannot be exported

This message indicates that you are trying to export a function or procedure that is tagged with the local directive. You also, cannot export threadvars and you would receive this message if you try to do so.

3.1.2.1.68 E2071: This type cannot be initialized

File types (including type Text), and the type Variant cannot be initialized, that is, you cannot declare typed constants or initialized variables of these types.

```
program Produce;

var
  V: Variant = 0;

begin
end.
```

The example tries to declare an initialized variable of type Variant, which illegal.

```
program Solve;

var
  V: Variant;

begin
  V := 0;
end.
```

The solution is to initialize a normal variable with an assignment statement.

3.1.2.1.69 E2374: Cannot make unique type from %s

No further information is available for this error or warning.

3.1.2.1.70 E2223: $DENYPACKAGEUNIT '%s' cannot be put into a package

You are attempting to put a unit which was compiled with $DENYPACKAGEUNIT into a package. It is not possible to put a unit compiled with the $DENYPACKAGEUNIT direction into a package.

3.1.2.1.71 E2217: Published field '%s' not a class or interface type

An attempt has been made to publish a field in a class which is not a class nor interface type.

```
program Produce;

  type
    TBaseClass = class
    published
      x : Integer;
    end;
begin
end.
```

The program above generates an error because x is included in a published section, despite the fact that it is not of a type which can be published.

```
program Solve;
  type
    TBaseClass = class
      Fx : Integer;
    published
      property X : Integer read Fx write Fx;
    end;

begin
end.
```

To solve this problem, all fields which are not class nor interface types must be removed from the published section of a class. If it is a requirement that the field actually be published, then it can be accomplished by changing the field into a property, as was done in this example.

3.1.2.1.72 E2218: Published method '%s' contains an unpublishable type

This message is not used in dccil. The message applies only to Win32 compilations, where it indicates that a parameter or function result type in the method is not a publishable type.

3.1.2.1.73 E2278: Cannot take address of local symbol %s

This message occurs when you try to call a symbol from within a procedure or function that has been tagged with the local directive.

The local directive, which marks routines as unavailable for export, is platform-specific and has no effect in Windows programming.

On Linux, the local directive is used for routines that are compiled into a library but are not exported. This directive can be specified for standalone procedures and functions, but not for methods. A routine declared with local, for example,

```
function Contraband(I: Integer): Integer; local;
```

does not refresh the EBX register and hence

- cannot be exported from a library.
- cannot be declared in the interface section of a unit.
- cannot have its address taken or be assigned to a procedural-type variable.
- if it is a pure assembler routine, cannot be called from another unit unless the caller sets up EBX.

3.1.2.1.74 E2392: Can't generate required accessor method(s) for property %s.%s due to name conflict with existing symbol %s in the same scope

The CLR requires that property accessors be methods, not fields. The Delphi language allows you to specify fields as property accessors. The Delphi compiler will generate the necessary methods behind the scenes. CLS recommends a specific naming convention for property accessor methods: get_propname and set_propname. If the accessors for a property are not methods, or if the given methods do not match the CLS name pattern, the Delphi compiler will attempt to generate methods with CLS conforming names. If a method already exists in the class that matches the CLS name pattern, but it is not associated with the particular property, the compiler cannot generate a new property accessor method with the CLS name pattern.

If the given property's accessors are methods, name collisions prevent the compiler from producing a CLS conforming name, but does not prevent the property from being usable.

However, if a name conflict prevents the compiler from generating an accessor method for a field accessor, the property is not usable and you will receive this error.

3.1.2.1.75 E2126: Cannot BREAK, CONTINUE or EXIT out of a FINALLY clause

Because a FINALLY clause may be entered and exited through the exception handling mechanism or through normal program control, the explicit control flow of your program may not be followed. When the FINALLY is entered through the exception handling mechanism, it is not possible to exit the clause with BREAK, CONTINUE, or EXIT - when the finally clause is being executed by the exception handling system, control must return to the exception handling system.

```
program Produce;

  procedure A0;
  begin
    try
      (* try something that might fail *)
    finally
      break;
    end;
  end;

  begin
  end.
```

The program above attempts to exit the finally clause with a break statement. It is not legal to exit a FINALLY clause in this manner.

```
program Solve;

  procedure A0;
  begin
    try
      (* try something that might fail *)
    finally
    end;
  end;

  begin
  end.
```

The only solution to this error is to restructure your code so that the offending statement does not appear in the FINALLY clause.

3.1.2.1.76 W1018: Case label outside of range of case expression

You have provided a label inside a case statement which cannot be produced by the case statement control variable. -W

```
program Produce;
(*$WARNINGS ON*)

  type
    CompassPoints = (n, e, s, w, ne, se, sw, nw);
    FourPoints = n..w;

  var
    TatesCompass : FourPoints;

begin

   TatesCompass := e;
   case TatesCompass OF
   n:    Writeln('North');
   e:    Writeln('East');
   s:    Writeln('West');
   w:    Writeln('South');
   ne:   Writeln('Northeast');
   se:   Writeln('Southeast');
   sw:   Writeln('Southwest');
   nw:   Writeln('Northwest');
   end;
end.
```

It is not possible for a TatesCompass to hold all the values of the CompassPoints, and so several of the case labels will elicit errors.

```
program Solve;
(*$WARNINGS ON*)

  type
    CompassPoints = (n, e, s, w, ne, se, sw, nw);
    FourPoints = n..w;

  var
    TatesCompass : CompassPoints;

begin

   TatesCompass := e;
   case TatesCompass OF
   n:    Writeln('North');
   e:    Writeln('East');
   s:    Writeln('West');
   w:    Writeln('South');
   ne:   Writeln('Northeast');
   se:   Writeln('Southeast');
   sw:   Writeln('Southwest');
   nw:   Writeln('Northwest');
   end;
end.
```

After examining your code to determine what the intention was, there are two alternatives. The first is to change the type of the case statement's control variable so that it can produce all the case labels. The second alternative would be to remove any case labels that cannot be produced by the control variable. The first alternative is shown in this example.

3.1.2.1.77 E2326: Attribute '%s' can only be used once per target

This attribute can only be used once per target Attributes and their descendants may be declared with an AttributeUsage Attribute which describes how a custom Attribute may be used. If the use of an attribute violates AttributeUsage.allowmultiple then this error will be raised.

3.1.2.1.78 E2325: Attribute '%s' is not valid on this target

Attribute is not valid on this target. Attributes and their descendants may be declared with an AttributeUsage Attribute which describes how a custom Attribute may be used. If the use of an attribute violates AttributeUsage.validon property then this error will be raised. AttributeUsage.validon specifies the application element that this attribute may be applied to.

3.1.2.1.79 E2358: Class constructors not allowed in class helpers

No further information is available for this error or warning.

3.1.2.1.80 E2360: Class constructors cannot have parameters

No further information is available for this error or warning.

3.1.2.1.81 E2340: Metadata - Data too large

No further information is available for this error or warning.

3.1.2.1.82 E2343: Metadata - Primary key column may not allow the null value

No further information is available for this error or warning.

3.1.2.1.83 E2341: Metadata - Column cannot be changed

No further information is available for this error or warning.

3.1.2.1.84 E2342: Metadata - Too many RID or primary key columns, 1 is max

No further information is available for this error or warning.

3.1.2.1.85 E2329: Metadata - Error occured during a read

No further information is available for this error or warning.

3.1.2.1.86 E2330: Metadata - Error occured during a write

No further information is available for this error or warning.

3.1.2.1.87 E2334: Metadata - Old version error

No further information is available for this error or warning.

3.1.2.1.88 E2331: Metadata - File is read only

No further information is available for this error or warning.

3.1.2.1.89 E2339: Metadata - The importing scope is not compatible with the emitting scope

No further information is available for this error or warning.

3.1.2.1.90 E2332: Metadata - An ill-formed name was given

No further information is available for this error or warning.

3.1.2.1.91 E2337: Metadata - There isn't .CLB data in the memory or stream

No further information is available for this error or warning.

3.1.2.1.92 E2338: Metadata - Database is read only

No further information is available for this error or warning.

3.1.2.1.93 E2335: Metadata - A shared mem open failed to open at the originally

No further information is available for this error or warning.

3.1.2.1.94 E2336: Metadata - Create of shared memory failed. A memory mapping of the same name already exists

No further information is available for this error or warning.

3.1.2.1.95 E2344: Metadata - Data too large

No further information is available for this error or warning.

3.1.2.1.96 E2333: Metadata - Data value was truncated

No further information is available for this error or warning.

3.1.2.1.97 F2047: Circular unit reference to '%s'

One or more units use each other in their interface parts.

As the compiler has to translate the interface part of a unit before any other unit can use it, the compiler must be able to find a compilation order for the interface parts of the units.

Check whether all the units in the uses clauses are really necessary, and whether some can be moved to the implementation part of a unit instead.

```
unit A;
interface
uses B;           (*A uses B, and B uses A*)
implementation
end.

unit B;
interface
uses A;
implementation
end.
```

The problem is caused because A and B use each other in their interface sections.

```
unit A;
interface
```

```
uses B;          (*Compilation order: B.interface, A, B.implementation*)
implementation
end.

unit B;
interface
implementation
uses A;          (*Moved to the implementation part*)
end.
```

You can break the cycle by moving one or more uses to the implementation part.

3.1.2.1.98 E2123: PROCEDURE, FUNCTION, PROPERTY, or VAR expected

The tokens that follow "class" in a member declaration inside a class type are limited to procedure, function, var, and property.

3.1.2.1.99 E2061: Local class or interface types not allowed

Corresponds to object_local in previous compilers. Class and interface types cannot be declared inside a procedure body.

3.1.2.1.100 E2435: Class member declarations not allowed in anonymous record or local record type

Record types that are declared in local scopes or declared in-place in variable declarations can only contain field declarations. For advanced features in record types (such as methods, properties, and nested types), the record type must be an explicitly declared global type.

3.1.2.1.101 E2060: Class and interface types only allowed in type section

Class or interface types must always be declared with an explicit type declaration in a type section. Unlike record types, they cannot be anonymous.

The main reason for this is that there would be no way you could declare the methods of that type (since there is no type name).

Incorrect (attempting to declare a class type within a variable declaration):

```
program Produce;

var
  MyClass : class
    Field: Integer;
  end;

begin
end.
```

Correct:

```
program Solve;

type
  TMyClass = class
    Field: Integer;
  end;

var
  MyClass : TMyClass;

begin
```

```
end.
```

3.1.2.1.102 E2355: Class property accessor must be a class field or class static method

No further information is available for this error or warning.

3.1.2.1.103 E2128: %s clause expected, but %s found

The compiler was, due to the Delphi language syntax, expecting to find a clause1 in your program, but instead found clause2.

```
program Produce;

  type
    CharDesc = class
      vch : Char;

property Ch : Char;
    end;
end.
```

The first declaration of a property must specify a read and write clause, and since both are missing on the 'Ch' property, an error will result when compiling. In the case of properties, the original intention might have been to hoist a property defined in a base class to another visibility level - for example, from public to private. In this case, the most probable cause of the error is that the property name was not found in the base class. Make sure that you have spelled the property name correctly and that it is actually contained in one of the parent classes.

```
program Produce;

  type
    CharDesc = class
      vch : Char;

property Ch : Char read vch write vch;
    end;
end.
```

The solution is to ensure that all the proper clauses are specified, where required.

3.1.2.1.104 E2401: Failure loading .NET Framework %s: %08X

No further information is available for this error or warning.

3.1.2.1.105 x2421: Imported identifier '%s' conflicts with '%s' in '%s'

When importing type information from a .NET assembly, the compiler may encounter symbols that do not conform to CLS specifications. One example of this is case-sensitive versus case-insensitive identifiers. Another example is having a property in a class with the same name as a method or field in the same class. This error message indicates that same-named symbols were found in the same scope (members of the same class or interface) in an imported assembly and that only one of them will be accessible from Delphi syntax.

3.1.2.1.106 E2422: Imported identifier '%s' conflicts with '%s' in namespace '%s'

When importing type information from a .NET assembly, the compiler may encounter symbols that do not conform to CLS specifications. One example of this is case-sensitive versus case-insensitive identifiers. Another example is having a property in a class with the same name as a method or field in the same class. This error message indicates that same-named symbols were found in the same scope (members of the same class or interface) in an imported assembly and that only one of them will be accessible from Delphi syntax.

3.1.2.1.107 H2384: CLS: overriding virtual method '%s.%s' visibility (%s) must match base class '%s' (%s)

No further information is available for this error or warning.

3.1.2.1.108 E2431: for-in statement cannot operate on collection type '%s' because '%s' does not contain a member for '%s', or it is inaccessible

A for-in statement can only operate on the following collection types:

- Primitive types that the compiler recognizes, such as arrays, sets or strings
- Types that implement IEnumerable
- Types that implement the GetEnumerator pattern as documented in the Delphi Language Guide

Ensure that the specified type meets these requirements.

See Also

Declarations and Statements (see page 705)

3.1.2.1.109 W1024: Combining signed and unsigned types - widened both operands

To mathematically combine signed and unsigned types correctly the compiler must promote both operands to the next larger size data type and then perform the combination.

To see why this is necessary, consider two operands, an Integer with the value -128 and a Cardinal with the value 130. The Cardinal type has one more digit of precision than the Integer type, and thus comparing the two values cannot accurately be performed in only 32 bits. The proper solution for the compiler is to promote both these types to a larger, common, size and then to perform the comparison.

The compiler will only produce this warning when the size is extended beyond what would normally be used for calculating the result.

```
{$APPTYPE CONSOLE}
program Produce;
  var
    i : Integer;
    c : Cardinal;

begin
  i := -128;
  c := 130;
  WriteLn(i + c);
end.
```

In the example above, the compiler warns that the expression will be calculated at 64 bits rather than the supposed 32 bits.

3.1.2.1.110 E2008: Incompatible types

This error message occurs when the compiler expected two types to be compatible (meaning very similar), but in fact, they turned out to be different. This error occurs in many different situations - for example when a read or write clause in a property mentions a method whose parameter list does not match the property, or when a parameter to a standard procedure or function is of the wrong type.

This error can also occur when two units both declare a type of the same name. When a procedure from an imported unit has a parameter of the same-named type, and a variable of the same-named type is passed to that procedure, the error could occur.

```
unit unit1;
interface
  type
    ExportedType = (alpha, beta, gamma);

implementation
begin
end.

unit unit2;
interface
  type
    ExportedType = (alpha, beta, gamma);

  procedure ExportedProcedure(v : ExportedType);

implementation
  procedure ExportedProcedure(v : ExportedType);
  begin
  end;

begin
end.

program Produce;
uses unit1, unit2;

var
  A: array [0..9] of char;
  I: Integer;
  V : ExportedType;
begin
  ExportedProcedure(v);
  I:= Hi(A);
end.
```

The standard function Hi expects an argument of type Integer or Word, but we supplied an array instead. In the call to ExportedProcedure, V actually is of type unit1.ExportedType since unit1 is imported prior to unit2, so an error will occur.

```
unit unit1;
interface
  type
    ExportedType = (alpha, beta, gamma);

implementation
begin
end.

unit unit2;
interface
  type
    ExportedType = (alpha, beta, gamma);

  procedure ExportedProcedure(v : ExportedType);
```

```
implementation
  procedure ExportedProcedure(v : ExportedType);
  begin
  end;

begin
end.

program Solve;
uses unit1, unit2;
var
  A: array [0..9] of char;
  I: Integer;
  V : unit2.ExportedType;
begin
  ExportedProcedure(v);
  I:= High(A);
end.
```

We really meant to use the standard function High, not Hi. For the ExportedProcedure call, there are two alternative solutions. First, you could alter the order of the uses clause, but it could also cause similar errors to occur. A more robust solution is to fully qualify the type name with the unit which declared the desired type, as has been done with the declaration for V above.

3.1.2.1.111 E2009: Incompatible types: '%s'

The compiler has detected a difference between the declaration and use of a procedure.

```
program Produce;

  type
    ProcedureParm0 = procedure; stdcall;
    ProcedureParm1 = procedure(VAR x : Integer);

  procedure WrongConvention; register;
  begin
  end;

  procedure WrongParms(x, y, z : Integer);
  begin
  end;

  procedure TakesParm0(p : ProcedureParm0);
  begin
  end;

  procedure TakesParm1(p : ProcedureParm1);
  begin
  end;

begin
  TakesParm0(WrongConvention);
  TakesParm1(WrongParms);
end.
```

The call of 'TakesParm0' will elicit an error because the type 'ProcedureParm0' expects a 'stdcall' procedure, whereas 'WrongConvention' is declared with the 'register' calling convention. Similarly, the call of 'TakesParm1' will fail because the parameter lists do not match.

```
program Solve;

  type
    ProcedureParm0 = procedure; stdcall;
    ProcedureParm1 = procedure(VAR x : Integer);
  procedure RightConvention; stdcall;
  begin
  end;

  procedure RightParms(VAR x : Integer);
  begin
  end;

  procedure TakesParm0(p : ProcedureParm0);
  begin
  end;

  procedure TakesParm1(p : ProcedureParm1);
  begin
  end;

begin
  TakesParm0(RightConvention);
  TakesParm1(RightParms);
end.
```

The solution to both of these problems is to ensure that the calling convention or the parameter lists matches the declaration.

3.1.2.1.112 E2010: Incompatible types: '%s' and '%s'

This error message results when the compiler expected two types to be compatible (or similar), but they turned out to be different.

```
program Produce;

procedure Proc(I: Integer);
begin
end;

begin
  Proc( 22 / 7 ); (*Result of / operator is Real*)
end.
```

Here a C++ programmer thought the division operator / would give him an integral result - not the case in Delphi.

```
program Solve;

procedure Proc(I: Integer);
begin
end;

begin
  Proc( 22 div 7 ); (*The div operator gives result type Integer*)
end.
```

The solution in this case is to use the integral division operator div - in general, you have to look at your program very careful to decide how to resolve type incompatibilities.

3.1.2.1.113 W1023: Comparing signed and unsigned types - widened both operands

To compare signed and unsigned types correctly the compiler must promote both operands to the next larger size data type.

To see why this is necessary, consider two operands, a Shortint with the value -128 and a Byte with the value 130. The Byte type has one more digit of precision than the Shortint type, and thus comparing the two values cannot accurately be performed in only 8 bits. The proper solution for the compiler is to promote both these types to a larger, common, size and then to perform the comparison.

```
program Produce;
  var
    s : shortint;
    b : byte;

begin
  s := -128;
  b := 130;

  assert(b < s);
end.
```

3.1.2.1.114 W1021: Comparison always evaluates to False

The compiler has determined that the expression will always evaluate to False. This most often can be the result of a boundary test against a specific variable type, for example, a Integer against $80000000.

In versions of the Delphi compiler prior to 12.0, the hexadecimal constant $80000000 would have been a negative Integer value, but with the introduction of the int64 type, this same constant now becomes a positive int64 type. As a result, comparisons of this constant against Integer variables will no longer behave as they once did.

As this is a warning rather than an error, there is no standard method of addressing the problems: sometimes the warning can be ignored, sometimes the code must be rewritten.

```
program Produce;

  var
    i : Integer;
    c : Cardinal;

begin
  c := 0;
  i := 0;
  if c < 0 then
    WriteLn('false');

  if i >= $80000000 then
    WriteLn('false');
end.
```

Here the compiler determines that the two expressions will always be False. In the first case, a Cardinal, which is unsigned, can never be less than 0. In the second case, a 32-bit Integer value can never be larger than, or even equal to, an int64 value of $80000000.

3.1.2.1.115 W1022: Comparison always evaluates to True

The compiler has determined that the expression will always evaluate to true. This most often can be the result of a boundary test against a specific variable type, for example, a Integer against $80000000.

In versions of the CodeGear Pascal compiler prior to 12.0, the hexadecimal constant $80000000 would have been a negative Integer value, but with the introduction of the int64 type, this same constant now becomes a positive int64 type. As a result, comparisons of this constant against Integer variables will no longer behave as they once did.

As this is a warning rather than an error, there is no standard method of addressing the problems: sometimes the warning can be ignored, sometimes the code must be rewritten.

```
program Produce;

  var
    i : Integer;
    c : Cardinal;

begin
  c := 0;
  i := 0;
  if c >= 0 then
    WriteLn('true');

  if i < $80000000 then
    WriteLn('true');
end.
```

Here the compiler determines that the two expressions will always be true. In the first case, a Cardinal, which is unsigned, will always be greater or equal to 0. In the second case, a 32-bit Integer value will always be smaller than an int64 value of $80000000.

3.1.2.1.116 E2026: Constant expression expected

The compiler expected a constant expression here, but the expression it found turned out not to be constant.

```
program Produce;
const
  Message = 'Hello World!';
  WPosition = Pos('W', Message);
begin
end.
```

The call to Pos is not a constant expression to the compiler, even though its arguments are constants, and it could in principle be evaluated at compile time.

```
program Solve;
const
  Message = 'Hello World!';
  WPosition = 7;
begin
end.
```

So in this case, we just have to calculate the right value for WPosition ourselves.

3.1.2.1.117 E2192: Constants cannot be used as open array arguments

Open array arguments must be supplied with an actual array variable, a constructed array or a single variable of the argument's element type.

```
program Produce;

  procedure TakesArray(s : array of String);
  begin
  end;


begin TakesArray('Hello Error');
end.
```

The error is caused in this example because a string literal is being supplied when an array is expected. It is not possible to implicitly construct an array from a constant.

```
program Solve;

  procedure TakesArray(s : array of String);
  begin
  end;
```

```
begin TakesArray(['Hello Error']);
end.
```

The solution avoids the error because the array is explicitly constructed.

3.1.2.1.118 E2007: Constant or type identifier expected

This error message occurs when the compiler expects a type, but finds a symbol that is neither a constant (a constant could start a subrange type), nor a type identifier.

```
program Produce;
var
  c : ExceptionClass; (*ExceptionClass is a variable in System*)
begin
end.
```

Here, ExceptionClass is a variable, not a type.

```
program Solve;
program Produce;
var
  c : Exception; (*Exception is a type in SysUtils*)
begin
end.
```

You need to make sure you specify a type. Maybe the identifier is misspelled, or it is hidden by some other identifier, for example from another unit.

3.1.2.1.119 E2197: Constant object cannot be passed as var parameter

This error message is reserved.

3.1.2.1.120 E2177: Constructors and destructors not allowed in OLE automation section

You have incorrectly tried to put a constructor or destructor into the 'automated' section of a class declaration.

```
program Produce;

  type
    Base = class
    automated
      constructor HardHatBob;
      destructor  DemolitionBob;
    end;

  constructor Base.HardHatBob;
  begin
  end;

  destructor Base.DemolitionBob;
  begin
  end;

begin
end.
```

It is not possible to declare a class constructor or destruction in an OLE automation section. The constructor and destructor declarations in the above code will both elicit this error.

```
program Solve;

  type
    Base = class
      constructor HardHatBob;
      destructor  DemolitionBob;
    end;

  constructor Base.HardHatBob;
  begin
  end;

  destructor Base.DemolitionBob;
  begin
  end;

begin
end.
```

The only solution to this error is to move your declarations out of the automated section, as has been done in this example.

3.1.2.1.121 x1020: Constructing instance of '%s' containing abstract method '%s.%s'

The code you are compiling is constructing instances of classes which contain abstract methods.

```
program Produce;
(*$WARNINGS ON*)
(*$HINTS ON*)

  type
    Base = class
      procedure Abstraction; virtual; abstract;
    end;

  var
    b : Base;

begin
  b := Base.Create;
end.
```

An abstract procedure does not exist, so it becomes dangerous to create instances of a class which contains abstract procedures. In this case, the creation of 'b' is the cause of the warning. Any invocation of 'Abstraction' through the instance of 'b' created here would cause a runtime error. A hint will be issued that the value assigned to 'b' is never used.

```
program Solve;
(*$WARNINGS ON*)
(*$HINTS ON*)

  type
    Base = class
      procedure Abstraction; virtual;
    end;

  var
    b : Base;

  procedure Base.Abstraction;
  begin
  end;

begin
  b := Base.Create;
```

```
end.
```

One solution to this problem is to remove the abstract directive from the procedure declaration, as is shown here. Another method of approaching the problem would be to derive a class from Base and then provide a concrete version of Abstraction. A hint will be issued that the value assigned to 'b' is never used.

3.1.2.1.122 E2402: Constructing instance of abstract class '%s'

No further information is available for this error or warning.

3.1.2.1.123 E2437: Constant declarations not allowed in anonymous record or local record type

Record types that are declared in local scopes or declared in-place in variable declarations can only contain field declarations. For advanced features in record types (such as methods, properties, and nested types), the record type must be an explicitly declared global type.

3.1.2.1.124 E2241: C++ obj files must be generated (-jp)

Because of the language features used, standard C object files cannot be generated for this unit. You must generate C++ object files.

3.1.2.1.125 E2412: CREATE expected

No further information is available for this error or warning.

3.1.2.1.126 E2306: 'Self' is initialized more than once

An inherited constructor has been initialized multiple times.

3.1.2.1.127 E2304: 'Self' is uninitialized. An inherited constructor must be called

In Delphi for .NET, a constructor must always call an inherited constructor before it may access or initialize any inherited class members. The compiler generates an error if your constructor code does not call the inherited constructor (a valid situation in Delphi for Win32), but it is important to examine your constructors to make sure that you do not access any inherited class fields, directly or indirectly, before the call to the inherited constructor.

Note: A constructor can initialize fields from its own class, prior to calling the inherited constructor.

Example:

The class,

```
X=class
  constructor Create;
  end;
```

requires an inherited constructor in its Create method:

```
constructor X.Create;
begin
  inherited Create;
```

```
end;
```

3.1.2.1.128 E2305: 'Self' might not have been initialized

In Delphi for .NET, a constructor must always call an inherited constructor before it may access or initialize any inherited class members. The compiler generates an error if your constructor code does not call the inherited constructor (a valid situation in Delphi for Win32), but it is important to examine your constructors to make sure that you do not access any inherited class fields, directly or indirectly, before the call to the inherited constructor.

Note: A constructor can initialize fields from its own class, prior to calling the inherited constructor.

3.1.2.1.129 E2302: 'Self' is uninitialized. An inherited constructor must be called before accessing ancestor field '%s'

In Delphi for .NET, a constructor must always call an inherited constructor before it may access or initialize any inherited class members. The compiler generates an error if your constructor code does not call the inherited constructor (a valid situation in Delphi for Win32), but it is important to examine your constructors to make sure that you do not access any inherited class fields, directly or indirectly, before the call to the inherited constructor.

Note: A constructor can initialize fields from its own class, prior to calling the inherited constructor.

3.1.2.1.130 E2303: 'Self' is uninitialized. An inherited constructor must be called before calling ancestor method '%s'

In Delphi for .NET, a constructor must always call an inherited constructor before it may access or initialize any inherited class members. The compiler generates an error if your constructor code does not call the inherited constructor (a valid situation in Delphi for Win32), but it is important to examine your constructors to make sure that you do not access any inherited class fields, directly or indirectly, before the call to the inherited constructor.

Note: A constructor can initialize fields from its own class, prior to calling the inherited constructor.

3.1.2.1.131 E2286: Coverage library name is too long: %s

This message is not used in this product.

3.1.2.1.132 H2455: Narrowing given wide string constant lost information

Any character in a WideString constant with ordinal value greater than 127 may be replaced with "?" if the WideChar is not representable in the current locale codepage.

3.1.2.1.133 H2451: Narrowing given WideChar constant (#$%04X) to AnsiChar lost information

An AnsiChar can only represent the first 256 values in a WideChar, so the second byte of the WideChar is lost when converting it to an AnsiChar. You may wish to use WideChar instead of AnsiChar to avoid information loss.

3.1.2.1.134 E2238: Default value required for '%s'

When using default parameters a list of parameters followed by a type is not allowed; you must specify each variable and its default value individually.

```
program Produce;

  procedure p0(a, b : Integer = 151);
  begin
  end;

begin
end.
```

The procedure definitions shown above will cause this error since it declares two parameters with a default value.

```
program Solve;

  procedure p0(a : Integer; b : Integer = 151);
  begin
  end;

  procedure p1(a : Integer = 151; b : Integer = 151);
  begin
  end;

begin
end.
```

Depending on the desired result, there are different ways of approaching this problem. If only the last parameter is supposed to have the default value, then take the approach shown in the first example. If both parameters are supposed to have default values, then take the approach shown in the second example.

3.1.2.1.135 E2237: Parameter '%s' not allowed here due to default value

When using default parameters a list of parameters followed by a type is not allowed; you must specify each variable and its default value individually.

```
program Produce;

  procedure p0(a, b : Integer = 151);
  begin
  end;

begin
end.
```

The procedure definitions shown above will cause this error since it declares two parameters with a default value.

```
program Solve;

  procedure p0(a : Integer; b : Integer = 151);
  begin
  end;

  procedure p1(a : Integer = 151; b : Integer = 151);
  begin
  end;

begin
end.
```

Depending on the desired result, there are different ways of approaching this problem. If only the last parameter is supposed to have the default value, then take the approach shown in the first example. If both parameters are supposed to have default values, then take the approach shown in the second example.

3.1.2.1.136 E2132: Default property must be an array property

The default property which you have specified for the class is not an array property. Default properties are required to be array properties.

```
program Produce;

  type
    Base = class
      function GetV : Char;
      procedure SetV(x : Char);

      property Data : Char read GetV write SetV; default;
    end;

  function Base.GetV : Char;
  begin GetV := 'A';
  end;

  procedure Base.SetV(x : Char);
  begin
  end;

begin
end.
```

When specifying a default property, you must make sure that it conforms to the array property syntax. The 'Data' property in the above code specifies a 'Char' type rather than an array.

```
program Solve;

  type
    Base = class
      function GetV(i : Integer) : Char;
      procedure SetV(i : Integer; const x : Char);

      property Data[i : Integer] : Char read GetV write SetV; default;
    end;

  function Base.GetV(i : Integer) : Char;
  begin GetV := 'A';
  end;

  procedure Base.SetV(i : Integer; const x : Char);
  begin
  end;

begin
end.
```

By changing the specification of the offending property to an array, or by removing the 'default' directive, you can remove this error.

3.1.2.1.137 E2268: Parameters of this type cannot have default values

The default parameter mechanism incorporated into the Delphi compiler allows only simple types to be initialized in this manner. You have attempted to use a type that is not supported.

```
program Produce;
type
  ArrayType = array [0..1] of integer;
  procedure p1(proc : ArrayType = [1, 2]);
  begin
  end;
end.
```

Default parameters of this type are not supported in the Delphi language.

```
program solve;
type
  ArrayType = array [0..1] of integer;

  procedure p1(proc : ArrayType);
  begin
  end;

end.
```

The only way to eliminate this error is to remove the offending parameter assignment or to change the type of the parameter to one that can be initialized with a default value.

3.1.2.1.138 E2239: Default parameter '%s' must be by-value or const

Parameters which are given default values cannot be passed by reference.

```
program Produce;

  procedure p0(var x : Integer = 151);
  begin
  end;

begin
end.
```

Since the parameter x is passed by reference in this example, it cannot be given a default value.

```
program Solve;

  procedure p0(const x : Integer = 151);
  begin
  end;

begin
end.
```

In this solution, the by-reference parameter has been changed into a const parameter. Alternatively it could have been changed into a by-value parameter or the default value could have been removed.

3.1.2.1.139 E2131: Class already has a default property

You have tried to assign a default property to a class which already has defined a default property.

```
program Produce;

  type
    Base = class
      function GetV(i : Integer) : Char;
      procedure SetV(i : Integer; const x : Char);

      property Data[i : Integer] : Char read GetV write SetV; default;
      property Access[i : Integer] : Char read GetV write SetV; default;
    end;

  function Base.GetV(i : Integer) : Char;
  begin GetV := 'A';
  end;

  procedure Base.SetV(i : Integer; const x : Char);
  begin
  end;

begin
end.
```

The Access property in the code above attempts to become the default property of the class, but Data has already been specified as the default. There can be only one default property in a class.

```
program Solve;

  type
    Base = class
      function GetV(i : Integer) : Char;
      procedure SetV(i : Integer; const x : Char);

      property Data[i : Integer] : Char read GetV write SetV; default;
    end;

  function Base.GetV(i : Integer) : Char;
  begin GetV := 'A';
  end;

  procedure Base.SetV(i : Integer; const x : Char);
  begin
  end;

begin
end.
```

The solution is to remove the incorrect default property specifications from the program source.

3.1.2.1.140 E2146: Default values must be of ordinal, pointer or small set type

You have declared a property containing a default clause, but the type property type is incompatible with default values.

```
  program Produce;

    type
      VisualGauge = class
        pos : Single;
  property Position : Single read pos write pos default 0.0;
      end;

  begin
  end.
```

The program above creates a property and attempts to assign a default value to it, but since the type of the property does not allow default values, an error is output.

```
  program Produce;

    type
      VisualGauge = class
        pos : Integer;
  property Position : Integer read pos write pos default 0;
      end;

  begin
  end.
```

When this error is encountered, there are two easy solutions: the first is to remove the default value definition, and the second is to change the type of the property to one which allows a default value. Your program, however, may not be as simple to fix; consider when you have a set property which is too large - it is this case which will require you to carefully examine your program to determine the best solution to this problem.

3.1.2.1.141 F2087: System unit incompatible with trial version

You are using a trial version of the software. It is incompatible with the application you are trying to run.

3.1.2.1.142 E2144: Destination is inaccessible

The address to which you are attempting to put a value is inaccessible from within the IDE.

3.1.2.1.143 E2453: Destination cannot be assigned to

The integrated debugger has determined that your assignment is not valid in the current context.

3.1.2.1.144 E2290: Cannot mix destructors with IDisposable

The compiler will generate IDisposable support for a class that declares a destructor override named "Destroy". You cannot manually implement IDisposable and implement a destructor on the same class.

3.1.2.1.145 F2446: Unit '%s' is compiled with unit '%s' in '%s' but different version '%s' found

This error occurs if a unit must be recompiled to take in changes to another unit, but the source for the unit that needs recompilation is not found.

Note: This error message may be experienced when using inline functions. Expansion of an inline function exposes its implementation to all units that call the function. When a function is inline, modifications to that function must be reflected with a recompile of every unit that uses that function. This is true even if all of the modifications occur in the implementation

section. This is one way in which inlining can make your units more interdependent, requiring greater effort to maintain binary compatibility. This is of greatest importance to developers who distribute .dcu files without source code.

3.1.2.1.146 E2210: '%s' directive not allowed in in interface type

A directive was encountered during the parsing of an interface which is not allowed.

```
program Produce;
  type
    IBaseIntf = interface
    private
      procedure fnord(x, y, z : Integer);
    end;

begin
end.
```

In this example, the compiler gives an error when it encounters the private directive, as it is not allowed in interface types.

```
program Solve;
  type
    IBaseIntf = interface
```

```
      procedure fnord(x, y, z : Integer);
    end;

    TBaseClass = class (TInterfacedObject, IBaseIntf)
    private
      procedure fnord(x, y, z : Integer);
    end;

  procedure TBaseClass.fnord(x, y, z : Integer);
  begin
  end;
begin
end.
```

The only solution to this problem is to remove the offending directive from the interface definition. While interfaces do not actually support these directives, you can place the implementing method into the desired visibility section. In this example, placing the TBaseClass.fnord procedure into a private section should have the desired results.

3.1.2.1.147 E2228: A dispinterface type cannot have an ancestor interface

An interface type specified with dispinterface cannot specify an ancestor interface.

```
program Produce;

  type
    IBase = interface
    end;

    IExtend = dispinterface (IBase)
    ['{00000000-0000-0000-0000-000000000000}']

    end;

begin
end.
```

In the example above, the error is caused because IExtend attempts to specify an ancestor interface type.

```
program Solve;

  type
    IBase = interface
    end;

    IExtend = dispinterface
    ['{00000000-0000-0000-0000-000000000000}']

    end;

begin
end.
```

Generally there are two solutions when this error occurs: remove the ancestor interface declaration, or change the dispinterface into a regular interface type. In the example above, the former approach was taken.

3.1.2.1.148 E2230: Methods of dispinterface types cannot specify directives

Methods declared in a dispinterface type cannot specify any calling convention directives.

```
program Produce;

  type
    IBase = dispinterface
    ['{00000000-0000-0000-0000-000000000000}']
      procedure yamadama; register;
    end;

begin
end.
```

The error in the example shown here is that the method 'yamadama' attempts to specify the register calling convention.

```
program Solve;

  type
    IBase = dispinterface
    ['{00000000-0000-0000-0000-000000000000}']
      procedure yamadama;
    end;

begin
end.
```

Since no dispinterface method can specify calling convention directives, the only solution to this problem is to remove the offending directive, as shown in this example.

3.1.2.1.149 E2229: A dispinterface type requires an interface identification

When using dispinterface types, you must always be sure to include a GUID specification for them.

```
program Produce;

  type
    IBase = dispinterface
    end;

begin
end.
```

In the example shown here, the dispinterface type does not include a GUID specification, and thus causes the compiler to emit an error.

```
program Solve;

  type
    IBase = dispinterface
    ['{00000000-0000-0000-0000-000000000000}']

    end;

begin
end.
```

Ensuring that each dispinterface has a GUID associated with it will cause this error to go away.

3.1.2.1.150 E2183: Dispid clause only allowed in OLE automation section

A dispid has been given to a property which is not in an automated section.

```
program Produce;

  type
     Base = class
       v : integer;
       procedure setV(x : integer);
       function getV : integer;
       property Value : integer read getV write setV dispid 151;
```

```
    end;

  procedure Base.setV(x : integer);
  begin v := x;
  end;

  function Base.getV : integer;
  begin getV := v;
  end;

begin
end.
```

This program attempts to set the dispid for an OLE automation object, but the property has not been declared in an automated section.

```
program Solve;

  type
    Base = class
      v : integer;
      procedure setV(x : integer);
      function getV : integer;
    automated
      property Value : integer read getV write setV dispid 151;
    end;

  procedure Base.setV(x : integer);
  begin v := x;
  end;

  function Base.getV : integer;
  begin getV := v;
  end;

begin
end.
```

To solve the error, you can either remove the dispid clause from the property declaration, or move the property declaration into an automated section.

3.1.2.1.151 E2274: property attribute 'label' cannot be used in dispinterface

You have added a label to a property defined in a dispinterface, but this is disallowed by the language definition.

```
program Problem;

  type
    T0 = dispinterface
      ['{15101510-1510-1510-1510-151015101510}']
      function R : Integer;
      property value : Integer label 'Key';
    end;

begin
end.
```

Here an attempt is made to use a label attribute on a dispinterface property.

```
program Solve;

  type
    T0 = dispinterface
      ['{15101510-1510-1510-1510-151015101510}']
      function R : Integer;
      property value : Integer;
    end;

begin
end.
```

The only solution to this problem is to remove label attribute from the property definition.

3.1.2.1.152 E2080: Procedure DISPOSE needs destructor

This error message is issued when an identifier given in the parameter list to Dispose is not a destructor.

```
program Produce;

type
  PMyObject = ^TMyObject;
  TMyObject = object
  F: Integer;
  constructor Init;
  destructor Done;
  end;

constructor TMyObject.Init;
begin
  F := 42;
end;

destructor TMyObject.Done;
begin
end;

var
  P: PMyObject;

begin
  New(P, Init);
  (*...*)
  Dispose(P, Init);        (*<-- Error message here*)
end.
```

In this example, we passed the constructor to Dispose by mistake.

```
program Solve;

type
  PMyObject = ^TMyObject;
  TMyObject = object
  F: Integer;
  constructor Init;
  destructor Done;
  end;

constructor TMyObject.Init;
begin
  F := 42;
end;

destructor TMyObject.Done;
begin
end;

var
  P: PMyObject;

begin
```

```
  New(P, Init);
  Dispose(P, Done);
end.
```

The solution is to either pass a destructor to Dispose, or to eliminate the second argument.

3.1.2.1.153 E2414: Disposed_ cannot be declared in classes with destructors

Disposed_ cannot be declared in classes with destructors. If a class implements the IDispose interface the compiler generates a field called Disposed_ to determine whether or not the IDispose.Dispose method has already been called.

3.1.2.1.154 E2098: Division by zero

The compiler has detected a constant division by zero in your program.

Check your constant expressions and respecify them so that a division by zero error will not occur.

3.1.2.1.155 E2293: Cannot have both a DLLImport attribute and an external or calling convention directive

The compiler emits DLLImport attributes internally for external function declarations. This error is raised if you declare your own DLLImport attribute on a function and use the external name clause on the function.

3.1.2.1.156 E2027: Duplicate tag value

This error message is given when a constant appears more than once in the declaration of a variant record.

```
program Produce;
type
  VariantRecord = record
    case Integer of
    0: (IntField: Integer);
    0: (RealField: Real);     (*<-- Error message here*)
  end;

begin
end.
    program Solve;
type
  VariantRecord = record
    case Integer of
    0: (IntField: Integer);
    1: (RealField: Real);
  end;

begin
end.
```

3.1.2.1.157 E2399: Namespace conflicts with unit name '%s'

No further information is available for this error or warning.

3.1.2.1.158 E2030: Duplicate case label

This error message occurs when there is more than one case label with a given value in a case statement.

```
program Produce;

function DigitCount(I: Integer): Integer;
begin
  case Abs(I) of
  0:                  DigitCount := 1;
  0        ..9:         DigitCount := 1;   (*<-- Error message here*)
  10       ..99:        DigitCount := 2;
  100      ..999:       DigitCount := 3;
  1000     ..9999:      DigitCount := 4;
  10000    ..99999:     DigitCount := 5;
  100000   ..999999:    DigitCount := 6;
  1000000  ..9999999:   DigitCount := 7;
  10000000 ..99999999:  DigitCount := 8;
  100000000..999999999: DigitCount := 9;
  else                  DigitCount := 10;
  end;
end;

begin
  Writeln( DigitCount(12345) );
end.
```

Here we did not pay attention and mentioned the case label 0 twice.

```
program Solve;

function DigitCount(I: Integer): Integer;
begin
  case Abs(I) of
  0        ..9:         DigitCount := 1;
  10       ..99:        DigitCount := 2;
  100      ..999:       DigitCount := 3;
  1000     ..9999:      DigitCount := 4;
  10000    ..99999:     DigitCount := 5;
  100000   ..999999:    DigitCount := 6;
  1000000  ..9999999:   DigitCount := 7;
  10000000 ..99999999:  DigitCount := 8;
  100000000..999999999: DigitCount := 9;
  else                  DigitCount := 10;
  end;
end;

begin
  Writeln( DigitCount(12345) );
end.
```

In general, the problem might not be so easy to spot when you have symbolic constants and ranges of case labels - you might have to write down the real values of the constants to find out what is wrong.

3.1.2.1.159 W1029: Duplicate %s '%s' with identical parameters will be inacessible from C++

An object file is being generated and Two, differently named, constructors or destructors with identical parameter lists have been created; they will be inaccessible if the code is translated to an HPP file because constructor and destructor names are converted to the class name. In C++ these duplicate declarations will appear to be the same function.

```
unit Produce;
interface
  type
    Base = class
      constructor ctor0(a, b, c : integer);
      constructor ctor1(a, b, c : integer);
    end;
```

```
implementation
constructor Base.ctor0(a, b, c : integer);
begin
end;

constructor Base.ctor1(a, b, c : integer);
begin
end;

begin
end.
```

As can be seen in this example, the two constructors have the same signature and thus, when the file is compiled with one of the -j options, will produce this warning.

```
unit Solve;
interface
  type
    Base = class
      constructor ctor0(a, b, c : integer);
      constructor ctor1(a, b, c : integer; dummy : integer = 0);
    end;

implementation
constructor Base.ctor0(a, b, c : integer);
begin
end;

constructor Base.ctor1(a, b, c : integer; dummy : integer);
begin
end;

begin
end.
```

A simple method to solve this problem is to change the signature of one of constructors, for example, to add an extra parameter. In the example above, a default parameter has been added to ctor1. This method of approaching this error has the benefit that Delphi code using ctor1 does not need to be changed. C++ code, on the other hand, will have to specify the extra parameter to allow the compiler to determine which constructor is desired.

3.1.2.1.160 E2180: Dispid '%d' already used by '%s'

An attempt to use a dispid which is already assigned to another member of this class.

```
program Produce;

  type
    Base = class
      v : Integer;
      procedure setV(x : Integer);
      function getV : Integer;
    automated
      property Value : Integer read getV write setV dispid 151;
      property SecondValue : Integer read getV write setV dispid 151;
    end;

  procedure Base.setV(x : Integer);
  begin v := x;
  end;

  function Base.getV : Integer;
  begin getV := v;
  end;
```

```
begin
end.
```

Each automated property's dispid must be unique, thus SecondValue is in error.

```
program Solve;

  type
    Base = class
      v : Integer;
      procedure setV(x : Integer);
      function getV : Integer;
    automated
      property Value : Integer read getV write setV dispid 151;
      property SecondValue : Integer read getV write setV dispid 152;
    end;

  procedure Base.setV(x : Integer);
  begin v := x;
  end;

  function Base.getV : Integer;
  begin getV := v;
  end;

begin
end.
```

Giving a unique dispid to SecondValue will remove the error.

3.1.2.1.161 E2301: Method '%s' with identical parameters and result type already exists

Within a class, you cannot publish multiple overloaded methods with the same name. Maintenance of runtime type information requires a unique name for each published member.

```
type
    TSomeClass = class
    published
    function Func(P: Integer): Integer;
    function Func(P: Boolean): Integer;   // error
```

3.1.2.1.162 E2257: Duplicate implements clause for interface '%s'

The compiler has encountered two different property declarations which claim to implement the same interface. An interface may be implemented by only one property.

```
program Produce;
type
  IMyInterface = interface
  end;

  TMyClass = class(TInterfacedObject, IMyInterface)
    FMyInterface: IMyInterface;
    property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
    property OtherInterface: IMyInterface read FMyInterface implements IMyInterface;
  end;
end.
```

Both MyInterface and OtherInterface attempt to implement IMyInterface. Only one property may implement the chosen interface.

The only solution in this case is to remove one of the offending implements clauses.

3.1.2.1.163 E2447: Duplicate symbol '%s' defined in namespace '%s' by '%s' and '%s'

This error occurs when symbols from separate units are combined into a common namespace, and the same symbol name is in both units. In previous versions of Delphi, these units may have compiled without error, because symbol scope was defined by the unit alone. In RAD Studio, units must be inserted into namespaces when generating the IL metadata. This may cause separate units to be be combined into a single namespace.

To resolve this problem, you may wish to rename one of the symbols in the two units, alias one of the symbols to the other, or change the unit names so that they do not contribute to the same namespace.

3.1.2.1.164 E2140: Duplicate message method index

You have specified an index for a dynamic method which is already used by another dynamic method.

```
program Produce;

  type
    Base = class
      procedure First(VAR x : Integer); message 151;
      procedure Second(VAR x : Integer); message 151;
    end;

  procedure Base.First(VAR x : Integer);
  begin
  end;

  procedure Base.Second(VAR x : Integer);
  begin
  end;

begin
end.
```

The declaration of 'Second' attempts to reuse the same message index which is used by 'First'; this is illegal.

```
program Solve;

  type
    Base = class
      procedure First(VAR x : Integer); message 151;
      procedure Second(VAR x : Integer); message 152; (*change to unique index*)
    end;

    Derived = class (Base)
      procedure First(VAR x : Integer); override; (*override base class behavior*)
    end;

  procedure Base.First(VAR x : Integer);
  begin
  end;

  procedure Base.Second(VAR x : Integer);
  begin
  end;

  procedure Derived.First(VAR x : Integer);
  begin
  end;

begin
```

```
end.
```

There are two straightforward solutions to this problem. First, if you really do not need to use the same message value, you can change the message number to be unique. Alternatively, you could derive a new class from the base and override the behavior of the message handler declared in the base class. Both options are shown in the above example.

3.1.2.1.165 E2252: Method '%s' with identical parameters already exists

A method with an identical signature already exists in the data type.

```
program Produce;

  type
    t0 = class
      procedure f0(a : integer); overload;
      procedure f0(a : integer); overload;
    end;

procedure T0.f0(a : integer);
begin
end;

begin
end.
```

The error is produced here because there are two overloaded declarations for the same procedure.

```
program Solve;

  type
    t0 = class
      procedure f0(a : integer); overload;
      procedure f0(a : char); overload;
    end;

procedure T0.f0(a : integer);
begin
end;

procedure T0.f0(a : char);
begin
end;

begin
end.
```

There are different approaches to resolving this error. One approach is to remove the redundant declaration of the procedure. Another approach, taken here, is to change the parameter type of the duplicate declarations so that it creates a unique version of the overloaded procedure.

3.1.2.1.166 E2266: Only one of a set of overloaded methods can be published

Only one member of a set of overloaded functions may be published because the RTTI generated for procedures only contains the name.

```
(*$M+*)
(*$APPTYPE CONSOLE*)
program Produce;
type
  Base = class
  published
    procedure p1(a : integer); overload;
    procedure p1(a : boolean); overload;
  end;

  Extended = class (Base)
    procedure e1(a : integer); overload;
    procedure e1(a : boolean); overload;
  end;

  procedure Base.p1(a : integer);
  begin
  end;

  procedure Base.p1(a : boolean);
  begin
  end;

  procedure Extended.e1(a : integer);
  begin
  end;

  procedure Extended.e1(a : boolean);
  begin
  end;

end.
```

In the example shown here, both overloaded p1 functions are contained in a published section, which is not allowed.

Further, since the $M+ state is used, the Extended class starts with published visibility, thus the error will also appear for this class also.

```
(*$M+*)
(*$APPTYPE CONSOLE*)
program Solve;
type
  Base = class
  public
    procedure p1(a : integer); overload;
  published
    procedure p1(a : boolean); overload;
  end;

  Extended = class (Base)
  public
    procedure e1(a : integer); overload;
    procedure e1(a : boolean); overload;
  end;

  procedure Base.p1(a : integer);
  begin
  end;

  procedure Base.p1(a : boolean);
  begin
  end;

  procedure Extended.e1(a : integer);
  begin
  end;

  procedure Extended.e1(a : boolean);
  begin
  end;

end.
```

The solution here is to ensure that no more than one member of a set of overloaded function appears in a published section. The easiest way to achieve this is to change the visibility to public, protected or private; whichever is most appropriate.

3.1.2.1.167 E2285: Duplicate resource id: type %d id %d

A resource linked into the project has the same type and name, or same type and resource ID, as another resource linked into the project. (In Delphi, duplicate resources are ignored with a warning. In Kylix, duplicates cause an error.)

3.1.2.1.168 E2407: Duplicate resource identifier %s found in unit %s(%s) and %s(%s)

No further information is available for this error or warning.

3.1.2.1.169 E2284: Duplicate resource name: type %d '%s'

A resource linked into the project has the same type and name, or same type and resource ID, as another resource linked into the project. (In Delphi, duplicate resources are ignored with a warning. In Kylix, duplicates cause an error.)

3.1.2.1.170 E2429: Duplicate implementation for 'set of %s' in this scope

To avoid this error, declare an explicit set type identifier instead of using in-place anonymous set expressions.

3.1.2.1.171 W1051: Duplicate symbol names in namespace. Using '%s.%s' found in %s. Ignoring duplicate in %s

No further information is available for this error or warning.

3.1.2.1.172 E2413: Dynamic array type needed

No further information is available for this error or warning.

3.1.2.1.173 E2178: Dynamic methods and message handlers not allowed in OLE automation section

You have incorrectly put a dynamic or message method into an 'automated' section of a class declaration.

```
program Produce;

  type
    Base = class
    automated
      procedure DynaMethod; dynamic;
      procedure MessageMethod(VAR msg : Integer); message 151;
    end;

    procedure Base.DynaMethod;
    begin
    end;

    procedure Base.MessageMethod;
    begin
    end;

begin
```

```
end.
```

It is not possible to have a dynamic or message method declaration in an OLE automation section of a class. As such, the two method declarations in the above program both produce errors.

```
program Solve;

  type
    Base = class
      procedure DynaMethod; dynamic;
      procedure MessageMethod(VAR msg : Integer); message 151;
    end;

    procedure Base.DynaMethod;
    begin
    end;

    procedure Base.MessageMethod;
    begin
    end;

begin
end.
```

There are several ways to remove this error from your program. First, you could move any declaration which produces this error out of the automated section, as has been done in this example. Alternatively, you could remove the dynamic or message attributes of the method; of course, removing these attributes will not provide you with the desired behavior, but it will remove the error.

3.1.2.1.174 E2378: Error while converting resource %s

No further information is available for this error or warning.

3.1.2.1.175 E2385: Error while signing assembly

No further information is available for this error or warning.

3.1.2.1.176 E2125: EXCEPT or FINALLY expected

The compiler was expecting to find a FINALLY or EXCEPT keyword, during the processing of exception handling code, but did not find either.

```
program Produce;

begin
  try
  end;
end.
```

In the code above, the 'except' or 'finally' clause of the exception handling code is missing, so the compiler will issue an error.

```
program Solve;

begin
  try
  except
  end;
end.
```

By adding the missing clause, the compiler will be able to complete the compilation of the code. In this case, the 'except' clause will easily allow the program to finish.

3.1.2.1.177 E2029: %s expected but %s found

This error message appears for syntax errors. There is probably a typo in the source, or something was left out. When the error occurs at the beginning of a line, the actual error is often on the previous line.

```
program Produce;
var
  I: Integer
begin               (*<-- Error message here: ';' expected but 'BEGIN' found*)
end.
```

After the type Integer, the compiler expects to find a semicolon to terminate the variable declaration. It does not find the semicolon on the current line, so it reads on and finds the 'begin' keyword at the start of the next line. At this point it finally knows something is wrong...

```
program Solve;
var
  I: Integer;       (*Semicolon was missing*)
begin
end.
```

In this case, just the semicolon was missing - a frequent case in practice. In general, have a close look at the line where the error message appears, and the line above it to find out whether something is missing or misspelled.

3.1.2.1.178 E2191: EXPORTS allowed only at global scope

An EXPORTS clause has been encountered in the program source at a non-global scope.

```
program Produce;

  procedure ExportedProcedure;
  exports ExportedProcedure;
  begin
  end;

begin
end.
```

It is not allowed to have an EXPORTS clause anywhere but a global scope.

```
program Solve;

  procedure ExportedProcedure;
  begin
  end;

exports ExportedProcedure;
begin
end.
```

The solution is to ensure that your EXPORTS clause is at a global scope and textually follows all procedures named in the clause. As a general rule, EXPORTS clauses are best placed right before the source file's initialization code.

3.1.2.1.179 E2143: Expression has no value

You have attempted to assign the result of an expression, which did not produce a value, to a variable.

3.1.2.1.180 E2353: Cannot extend sealed class '%s'

The sealed modifier is used to prevent inheritance (and thus extension) of a class.

3.1.2.1.181 E2078: Procedure FAIL only allowed in constructor

The standard procedure Fail can only be called from within a constructor - it is illegal otherwise.

3.1.2.1.182 E2169: Field definition not allowed after methods or properties

You have attempted to add more fields to a class after the first method or property declaration has been encountered. You must place all field definitions before methods and properties.

```
program Produce;

  type
    Base = class
      procedure FirstMethod;
      a : Integer;
    end;


  procedure Base.FirstMethod;
  begin
  end;

begin
end.
```

The declaration of 'a' after 'FirstMethod' will cause an error.

```
program Solve;

  type
    Base = class
      a : Integer;
      procedure FirstMethod;
    end;


  procedure Base.FirstMethod;
  begin
  end;

begin
end.
```

To solve this error, it is normally sufficient to move all field definitions before the first field or property declaration.

3.1.2.1.183 E2175: Field definition not allowed in OLE automation section

You have tried to place a field definition in an OLE automation section of a class declaration. Only properties and methods may be declared in an 'automated' section.

```
program Produce;

  type
    Base = class
    automated
      i : Integer;
    end;

begin
end.
```

The declaration of 'i' in this class will cause the compile error.

```
program Solve;

  type
    Base = class
      i : Integer;
    automated
    end;

begin
end.
```

Moving the declaration of 'i' out of the automated section will vanquish the error.

3.1.2.1.184 E2124: Instance member '%s' inaccessible here

You are attempting to reference a instance member from within a class procedure.

```
program Produce;

  type
    Base = class
      Title : String;

      class procedure Init;
    end;

  class procedure Base.Init;
  begin
    Self.Title := 'Does not work';
    Title := 'Does not work';
  end;

begin
end.
```

Class procedures do not have an instance pointer, so they cannot access any methods or instance data of the class.

```
program Solve;

  type
    Base = class
      Title : String;

      class procedure Init;
    end;

  class procedure Base.Init;
  begin
  end;

begin
end.
```

The only solution to this error is to not access any member data or methods from within a class method.

3.1.2.1.185 E2209: Field declarations not allowed in interface type

An interface has been encountered which contains definitions of fields; this is not permitted.

```
program Produce;
  type
    IBaseIntf = interface
      FVar : Integer;
      property Value : Integer read FVar write FVar;
    end;

begin
end.
```

The desire above is to have a property which has a value associated with it. However, as interfaces can have no fields, this idea will not work.

```
program Solve;
    IBaseIntf = interface
      function Reader : Integer;
      procedure Writer(a : Integer);
      property Value : Integer read Reader write Writer;
    end;

begin
end.
```

An elegant solution to the problem described above is to declare getter and setter procedures for the property. In this situation, any class implementing the interface must provide a method which will be used to access the data of the class.

3.1.2.1.186 x2044: Chmod error on '%s'

The file permissions are not properly set on a file. See the chmod man page for more information.

3.1.2.1.187 x2043: Close error on '%s'

The compiler encountered an error while closing an input or output file.

This should rarely happen. If it does, the most likely cause is a full or bad disk.

3.1.2.1.188 F2039: Could not create output file '%s'

The compiler could not create an output file. This can be a compiled unit file (.dcu), an executable file, a map file or an object file.

Most likely causes are a nonexistent directory or a write protected file or disk.

3.1.2.1.189 x2141: Bad file format: '%s'

The compiler state file has become corrupted. It is not possible to reload the previous compiler state.

Delete the corrupt file.

3.1.2.1.190 E2288: File name too long (exceeds %d characters)

A file path specified in the compiler options exceeds the compiler's file buffer length.

3.1.2.1.191 x1026: File not found: '%s'

This error message occurs when the compiler cannot find an input file. This can be a source file, a compiled unit file (.dcuil file), an include, an object file or a resource file.

Check the spelling of the name and the relevant search path.

```
program Produce;
uses Borland.Vcl.SysUtilss;       (*<-- Error message here*)
begin
end.
    program Solve;
uses Borland.Vcl.SysUtils;      (*Fixed typo*)
begin
end.
```

For a .dcuil file, failure to set the unit/library path for the compiler is a likely cause of this message. The only solution is to make sure the named unit can be found along the library path.

3.1.2.1.192 F1027: Unit not found: '%s' or binary equivalents (%s)

This error message occurs when the compiler cannot find a referenced unit (.dcuil) file.

Check the spelling of the referenced file name and the relevant search path.

3.1.2.1.193 x2041: Read error on '%s'

The compiler encountered a read error on an input file.

This should never happen - if it does, the most likely cause is corrupt data.

3.1.2.1.194 F2040: Seek error on '%s'

The compiler encountered a seek error on an input or output file.

This should never happen - if it does, the most likely cause is corrupt data.

3.1.2.1.195 E2002: File type not allowed here

File types are not allowed as value parameters and as the base type of a file type itself. They are also not allowed as function return types, and you cannot assign them - those errors will however produce a different error message.

```
program Produce;

procedure WriteInteger(T: Text; I: Integer);
begin
  Writeln(T, I);
end;

begin
end.
```

In this example, the problem is that T is value parameter of type Text, which is a file type. Recall that whatever gets written to a value parameter has no effect on the caller's copy of the variable - declaring a file as a value parameter therefore makes little sense.

```
program Solve;

procedure WriteInteger(var T: Text; I: Integer);
begin
  Writeln(T, I);
end;

begin
```

```
end.
```

Declaring the parameter as a var parameter solves the problem.

3.1.2.1.196 x2042: Write error on '%s'

The compiler encountered a write error while writing to an output file.

Most likely, the output disk is full.

3.1.2.1.197 E2351: Final methods must be virtual or dynamic

No further information is available for this error or warning.

3.1.2.1.198 E2155: Type '%s' needs finalization - not allowed in file type

Certain types are treated specially by the compiler on an internal basis in that they must be correctly finalized to release any resources that they might currently own. Because the compiler cannot determine what type is actually stored in a record's variant section at runtime, it is not possible to guarantee that these special data types are correctly finalized.

```
program Produce;

  type
    Data = record
      name : string;
    end;

  var
    inFile : file of Data;

begin
end.
```

String is one of those data types which need finalization, and as such they cannot be stored in a File type.

```
program Solve;

  type
    Data = record
      name : array [1..25] of Char;
    end;

  var
    inFile : file of Data;

begin
end.
```

One simple solution, for the case of String, is to redeclare the type as an array of characters. For other cases which require finalization, it becomes increasingly difficult to maintain a binary file structure with standard Pascal features, such as 'file of'. In these situations, it is probably easier to write specialized file I/O routines.

3.1.2.1.199 E2154: Type '%s' needs finalization - not allowed in variant record

Certain types are treated specially by the compiler on an internal basis in that they must be correctly finalized to release any resources that they might currently own. Because the compiler cannot determine what type is actually stored in a record's variant section at runtime, it is not possible to guarantee that these special data types are correctly finalized.

```
program Produce;
```

```
  type
    Data = record
      case kind:Char of
      'A': (str : String);
    end;

begin
end.
```

String is one of those types which requires special treatment by the compiler to correctly release the resources. As such, it is illegal to have a String in a variant section.

```
program Solve;

  type
    Data = record
      str : String;
    end;

begin
end.
```

One solution to this error is to move all offending declarations out of the variant section. Another solution would be to use pointer types (^String, for example) and manage the memory by yourself.

3.1.2.1.200 E2103: 16-Bit fixup encountered in object file '%s'

A 16-bit fixup has been found in one of the object modules linked to your program with the $L compiler directive. The compiler only supports 32 bit fixups in linked object modules.

Make sure that the linked object module is a 32 bit object module.

3.1.2.1.201 W1037: FOR-Loop variable '%s' may be undefined after loop

This warning is issued if the value of a for loop control variable is used after the loop.

You can only rely on the final value of a for loop control variable if the loop is left with a goto or exit statement.

The purpose of this restriction is to enable the compiler to generate efficient code for the for loop.

```
program Produce;
(*$WARNINGS ON*)

function Search(const A: array of Integer; Value: Integer): Integer;
begin
  for Result := 0 to High(A) do
    if A[Result] = Value then
      break;
end;

const
  A : array [0..9] of Integer = (1,2,3,4,5,6,7,8,9,10);

begin
  Writeln( Search(A,11) );
end.
```

In the example, the Result variable is used implicitly after the loop, but it is undefined if we did not find the value - hence the warning.

```
program Solve;
(*$WARNINGS ON*)
```

```
function Search(const A: array of Integer; Value: Integer): Integer;
begin
  for Result := 0 to High(A) do
    if A[Result] = Value then
      exit;
  Result := High(a)+1;
end;

const
  A : array [0..9] of Integer = (1,2,3,4,5,6,7,8,9,10);

begin
  Writeln( Search(A,11) );
end.
```

The solution is to assign the intended value to the control variable for the case where we don't exit the loop prematurely.

3.1.2.1.202 W1015: FOR-Loop variable '%s' cannot be passed as var parameter

An attempt has been made to pass the control variable of a FOR-loop to a procedure or function which takes a var parameter. This is a warning because the procedure which receives the control variable is able to modify it, thereby changing the semantics of the FOR-loop which issued the call.

```
program Produce;

  procedure p1(var x : Integer);
  begin
  end;

  procedure p0;
    var
      i : Integer;
  begin
    for i := 0 to 1000 do
      p1(i);
  end;

begin
end.
```

In this example, the loop control variable, i, is passed to a procedure which receives a var parameter. This is the main cause of the warning.

```
program Solve;
  procedure p1(x : Integer);
  begin
  end;

  procedure p0;
    var
      i : Integer;
  begin
    i := 0;
    while i <= 1000 do
      p1(i);
  end;


begin
end.
```

The easiest way to approach this problem is to change the parameter into a by-value parameter. However, there may be a good reason that it was a by-reference parameter in the begging, so you must be sure that this change of semantics in your program does not affect other code. Another way to approach this problem is change the for loop into an equivalent while loop, as is done in the above program.

3.1.2.1.203 E2032: For loop control variable must have ordinal type

The control variable of a for loop must have type Boolean, Char, WideChar, Integer, an enumerated type, or a subrange type.

```
program Produce;
var
  x: Real;
begin (*Plot sine wave*)
  for x := 0 to 2*pi/0.2 do                            (*<-- Error message here*)
    Writeln( '*': Round((Sin(x*0.2) + 1)*20) + 1 );
end.
```

The example uses a variable of type Real as the for loop control variable, which is illegal.

```
program Solve;
var
  x: Integer;
begin (*Plot sine wave*)
  for x := 0 to Round(2*pi/0.2) do
    Writeln( '*': Round((Sin(x*0.2) + 1)*20) + 1 );
end.
```

Instead, use the Integer ordinal type.

You may see this error if a FOR loop uses an Int64 or Variant control variable. This results from a limitation in the compiler which you can work around by replacing the FOR loop with a WHILE loop.

3.1.2.1.204 x1019: For loop control variable must be simple local variable

This error message is given when the control variable of a for statement is not a simple variable (but a component of a record, for instance), or if it is not local to the procedure containing the for statement.

For backward compatibility reasons, it is legal to use a global variable as the control variable - the compiler gives a warning in this case. Note that using a local variable will also generate more efficient code.

```
program Produce;

var
  I: Integer;
  A: array [0..9] of Integer;

procedure Init;
begin
  for I := Low(A) to High(a) do  (*<-- Warning given here*)
    A[I] := 0;
end;

begin
  Init;
end.
    program Solve;
var
  A: array [0..9] of Integer;

procedure Init;
var
  I: Integer;
begin
  for I := Low(A) to High(a) do
    A[I] := 0;
```

```
end;

begin
  Init;
end.
```

3.1.2.1.205 E2037: Declaration of '%s' differs from previous declaration

This error message occurs when the declaration of a procedure, function, method, constructor or destructor differs from its previous (forward) declaration.

This error message also occurs when you try to override a virtual method, but the overriding method has a different parameter list, calling convention etc.

```
program Produce;

type
  MyClass = class
    procedure Proc(Inx: Integer);
    function Func: Integer;
    procedure Load(const Name: string);
    procedure Perform(Flag: Boolean);
    constructor Create;
    destructor Destroy(Msg: string); override;      (*<-- Error message here*)
    class function NewInstance: MyClass; override;  (*<-- Error message here*)
  end;

procedure MyClass.Proc(Index: Integer);             (*<-- Error message here*)
begin
end;

function MyClass.Func: Longint;                     (*<-- Error message here*)
begin
end;

procedure MyClass.Load(Name: string);               (*<-- Error message here*)
begin
end;

procedure MyClass.Perform(Flag: Boolean); cdecl;    (*<-- Error message here*)
begin
end;

procedure MyClass.Create;                           (*<-- Error message here*)
begin
end;

function MyClass.NewInstance: MyClass;              (*<-- Error message here*)
begin
end;

begin
end.
```

As you can see, there are a number of reasons for this error message to be issued.

```
program Solve;

type
  MyClass = class
    procedure Proc(Inx: Integer);
    function Func: Integer;
    procedure Load(const Name: string);
    procedure Perform(Flag: Boolean);
    constructor Create;
    destructor Destroy; override;                   (*No parameters*)
    class function NewInstance: TObject; override;  (*Result type   *)
  end;

procedure MyClass.Proc(Inx: Integer);               (*Parameter name  *)
begin
end;

function MyClass.Func: Integer;                     (*Result type  *)
begin
end;

procedure MyClass.Load(const Name: string);         (*Parameter kind  *)
begin
end;

procedure MyClass.Perform(Flag: Boolean);           (*Calling convention*)
begin
end;

constructor MyClass.Create;                         (*constructor*)
begin
end;

class function MyClass.NewInstance: TObject;        (*class function*)
begin
end;

begin
end.
```

You need to carefully compare the 'previous declaration' with the one that causes the error to determine what is different between the two.

3.1.2.1.206 E2065: Unsatisfied forward or external declaration: '%s'

This error message appears when you have a forward or external declaration of a procedure or function, or a declaration of a method in a class or object type, and you don't define the procedure, function or method anywhere.

Maybe the definition is really missing, or maybe its name is just misspelled.

Note that a declaration of a procedure or function in the interface section of a unit is equivalent to a forward declaration - you have to supply the implementation (the body of the procedure or function) in the implementation section.

Similarly, the declaration of a method in a class or object type is equivalent to a forward declaration.

```
program Produce;

type
  TMyClass = class
  constructor Create;
  end;

function Sum(const a: array of Double): Double; forward;

function Summ(const a: array of Double): Double;
var
  i: Integer;
begin
  Result := 0.0;
  for i:= 0 to High(a) do
  Result := Result + a[i];
end;
```

```
begin
end.
```

The definition of Sum in the above example has an easy-to-spot typo.

```
program Solve;

type
  TMyClass = class
  constructor Create;
  end;

constructor TMyClass.Create;
begin
end;

function Sum(const a: array of Double): Double; forward;

function Sum(const a: array of Double): Double;
var
  i: Integer;
begin
  Result := 0.0;
  for i:= 0 to High(a) do
  Result := Result + a[i];
end;

begin
end.
```

The solution: make sure the definitions of your procedures, functions and methods are all there, and spelled correctly.

3.1.2.1.207 W1011: Text after final 'END.' - ignored by compiler

This warning is given when there is still source text after the final end and the period that constitute the logical end of the program. Possibly the nesting of begin-end is inconsistent (there is one end too many somewhere). Check whether you intended the source text to be ignored by the compiler - maybe it is actually quite important.

```
program Produce;

begin
end.

Text here is ignored by Delphi 16-bit - Delphi 32-bit or Kylix gives a warning.

    program Solve;

begin
end.
```

3.1.2.1.208 E2127: 'GOTO %s' leads into or out of TRY statement

The GOTO statement cannot jump into or out of an exception handling statement.

```
program Produce;

label 1, 2;

begin
  goto 1;
  try
1:
  except
    goto 2;
  end;
2:
end.
```

Both GOTO statements in the above code are incorrect. It is not possible to jump into, or out of, exception handling blocks.

The ideal solution to this problem is to avoid using GOTO statements altogether, however, if that is not possible you will have to perform more detailed analysis of the program to determine the correct course of action.

3.1.2.1.209 E2295: A class helper cannot introduce a destructor

Class helpers cannot declare destructors.

3.1.2.1.210 E2172: Necessary library helper function was eliminated by linker (%s)

The integrated debugger is attempting to use some of the compiler helper functions to perform the requested evaluate. The linker, on the other hand, determined that the helper function was not actually used by the program and it did not link it into the program.

1. Create a new application.
2. Place a button on the form.
3. Double click the button to be taken to the 'click' method.
4. Add a global variable, 'v', of type String to the interface section.
5. Add a global variable, 'p', of type PChar to the interface section.

The click method should read as:

1. procedure TForm1.Button1Click(Sender: TObject); begin v := 'Initialized'; p := NIL; v := 'Abid'; end;
2. Set a breakpoint on the second assignment to 'v'.
3. Compile and run the application.
4. Press the button.
5. After the breakpoint is reached, open the evaluator (Run|Evaluate/Watch).
6. Evaluate 'v'.
7. Move the cursor to the 'New Value' box.
8. Type in 'p'.
9. Choose Modify.

The compiler uses a special function to copy a PChar to a String. In order to reduce the size of the produced executable, if that special function is not used by the program, it is not linked in. In this case, there is no assignment of a PChar to a String, so it is eliminated by the linker.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
     v := 'Initialized';
     p := NIL;
     v := 'Abid';
     v := p;
end;
```

Adding the extra assignment of a PChar to a String will ensure that the linker includes the desired procedure in the program. Encountering this error during a debugging session is an indicator that you are using some language/environment functionality that was not needed in the original program.

3.1.2.1.211 W1010: Method '%s' hides virtual method of base type '%s'

You have declared a method which has the same name as a virtual method in the base class. Your new method is not a virtual method; it will hide access to the base's method of the same name.

```
program Produce;

  type
    Base = class
      procedure VirtuMethod; virtual;
      procedure VirtuMethod2; virtual;
    end;

    Derived = class (Base)
      procedure VirtuMethod;
      procedure VirtuMethod2;
    end;

  procedure Base.VirtuMethod;
  begin
  end;

  procedure Base.VirtuMethod2;
  begin
  end;

  procedure Derived.VirtuMethod;
  begin
  end;

  procedure Derived.VirtuMethod2;
  begin
  end;

begin
end.
```

Both methods declared in the definition of Derived will hide the virtual functions of the same name declared in the base class.

```
program Solve;

  type
    Base = class
      procedure VirtuMethod; virtual;
      procedure VirtuMethod2; virtual;
    end;

    Derived = class (Base)
      procedure VirtuMethod; override;
      procedure Virtu2Method;
    end;

  procedure Base.VirtuMethod;
  begin
  end;

  procedure Base.VirtuMethod2;
  begin
  end;

  procedure Derived.VirtuMethod;
  begin
  end;

  procedure Derived.Virtu2Method;
  begin
  end;

begin
end.
```

There are three alternatives to take when solving this warning.

First, you could specify override to make the derived class' procedure also virtual, and thus allowing inherited calls to still reference the original procedure.

Secondly, you could change the name of the procedure as it is declared in the derived class. Both methods are exhibited in this example.

Finally, you could add the reintroduce directive to the procedure declaration to cause the warning to be silenced for that particular method.

3.1.2.1.212 W1009: Redeclaration of '%s' hides a member in the base class

A property has been created in a class with the same name of a variable contained in one of the base classes. One possible, and not altogether apparent, reason for getting this error is that a new version of the base class hierarchy has been installed and it contains new member variables which have names identical to your properties' names. -W

```
(*$WARNINGS ON*)
program Produce;

  type
    Base = class
      v : integer;
    end;

    Derived = class (Base)
      ch : char;
      property v : char read ch write ch;
    end;

begin
end.
```

Derived.v overrides, and thus hides, Base.v; it will not be possible to access Base.v in any variable of type Derived without a typecast.

```
(*$WARNINGS ON*)
program Solve;
  type
    Base = class
      v : integer;
    end;

    Derived = class (Base)
      ch : char;
      property chV : char read ch write ch;
    end;

begin
end.
```

By changing the name of the property in the derived class, the error is alleviated.

3.1.2.1.213 E2198: %s cannot be applied to a long string

It is not possible to use the standard function HIGH with long strings. The standard function HIGH can, however, be applied to old-style short strings.

Since long strings dynamically size themselves, no analog to the HIGH function can be used.

This error can be caused if you are porting a 16-bit application, in which case the only string type available was a short string. If this is the case, you can turn off the long strings with the $H command line switch or the long-form directive $LONGSTRINGS.

If the HIGH was applied to a string parameter, but you still wish to use long strings, you could change the parameter type to 'openstring'.

```
program Produce;
  var
    i : Integer;
    s : String;

begin
  s := 'Hello Developers of the World';
  i := HIGH(s);
end.
```

In the example above, the programmer attempted to apply the standard function HIGH to a long string variable. This cannot be done.

```
(*$LONGSTRINGS OFF*)
program Solve;
  var
    i : Integer;
    s : String;

begin
  s := 'Hello Developers of the World';
  i := HIGH(s);
end.
```

By disabling long string parameters, the application of HIGH to a string variable is now allowed.

3.1.2.1.214 W1034: $HPPEMIT '%s' ignored

The $HPPEMIT directive can only appear after the unit header.

3.1.2.1.215 x1008: Integer and HRESULT interchanged

In Delphi, Integer, Longint, and HRESULT are compatible types, but in C++ the types are not compatible and will produce differently mangled C++ parameter names. To ensure that there will not be problems linking object files created with the Delphi compiler this message alerts you to possible problems. If you are compiling your source to an object file, this is an error. Otherwise, it is a warning.

```
program Produce;
  uses Windows;

  type
    I0 = interface (IUnknown)
      procedure p0(var x : Integer);
    end;

    C0 = class (TInterfacedObject, I0)
      procedure p0(var x : HRESULT);
    end;

  procedure C0.p0(var x : HRESULT);
  begin
  end;
```

```
begin
end.
```

The example shown here declares the interface and class methods differently. While they are equivalent in Delphi, they are not so in C++.

```
program Solve;

  uses Windows;

  type
    I0 = interface (IUnknown)
      procedure p0(var x : Integer);
    end;

    C0 = class (TInterfacedObject, I0)
      procedure p0(var x : Integer);
    end;

  procedure C0.p0(var x : Integer);
  begin
  end;

begin
end.
```

The easiest solution to this problem is to match the class-declared methods to be identical to the interface-declared methods.

3.1.2.1.216 W1000: Symbol '%s' is deprecated

The symbol is tagged (using the **deprecated** hint directive) as no longer current and is maintained for compatibility only. You should consider updating your source code to use another symbol, if possible.

The **$WARN** SYMBOL_DEPRECATED ON/OFF compiler directive turns on or off all warnings about the **deprecated** directive on symbols in the current unit.

3.1.2.1.217 E2372: Identifier expected

No further information is available for this error or warning.

3.1.2.1.218 W1003: Symbol '%s' is experimental

An "experimental" directive has been used on an identifier. "Experimental" indicates the presence of a class or unit which is incomplete or not fully tested.

3.1.2.1.219 W1001: Symbol '%s' is specific to a library

The symbol is tagged (using the **library** hint directive) as one that may not be available in all libraries. If you are likely to use different libraries, it may cause a problem.

The **$WARN** SYMBOL_LIBRARY ON/OFF compiler directive turns on or off all warnings about the **library** directive on symbols in the current unit.

3.1.2.1.220 W1002: Symbol '%s' is specific to a platform

The symbol is tagged (using the **platform** hint directive) as one that may not be available on all platforms. If you are writing cross-platform applications, it may cause a problem.

The **$WARN** SYMBOL_PLATFORM ON/OFF compiler directive turns on or off all warnings about the **platform** directive on symbols in the current unit.

3.1.2.1.221 E2004: Identifier redeclared: '%s'

The given identifier has already been declared in this scope - you are trying to reuse its name for something else.

```
program Tests;
var
  Tests: Integer;
begin
end.
```

Here the name of the program is the same as that of the variable - we need to change one of them to make the compiler happy.

```
program Tests;
var
  TestCnt: Integer;
begin
end.
```

3.1.2.1.222 E2003: Undeclared identifier: '%s'

The compiler could not find the given identifier - most likely it has been misspelled either at the point of declaration or the point of use. It might be from another unit that has not mentioned a uses clause.

```
program Produce;
var
  Counter: Integer;
begin
  Count := 0;
  Inc(Count);
  Writeln(Count);
end.
```

In the example, the variable has been declared as "Counter", but used as "Count". The solution is to either change the declaration or the places where the variable is used.

```
program Solve;
var
  Count: Integer;
begin
  Count := 0;
  Inc(Count);
  Writeln(Count);
end.
```

In the example we have chosen to change the declaration - that was less work.

3.1.2.1.223 E2427: Only one of IID or GuidAttribute can be specified

The GUID or IID of your interface can be specified using square brackets at the top of the interface declaration or using a .NET attribute before the interface declaration. You may use either style of GUID or IID declaration, but not both styles in the same type.

3.1.2.1.224 E2038: Illegal character in input file: '%s' (%s)

The compiler found a character that is illegal in Delphi programs.

This error message is caused most often by errors with string constants or comments.

```
program Produce;

begin
  Writeln("Hello world!");   (*<-- Error messages here*)
end.
```

Here a programmer fell back to C++ habits and quoted a string with double quotes.

```
program Solve;

begin
  Writeln('Hello world!');   (*Need single quotes in Delphi*)
end.
```

The solution is to use single quotes. In general, you need to delete the illegal character.

3.1.2.1.225 E2182: '%s' clause not allowed in OLE automation section

INDEX, STORED, DEFAULT and NODEFAULT are not allowed in OLE automation sections.

```
program Produce;

  type
     Base = class
       v : integer;
       procedure setV(x : integer);
       function getV : integer;
       automated
       property Value : integer read getV write setV nodefault;
     end;

  procedure Base.setV(x : integer);
  begin v := x;
  end;

  function Base.getV : integer;
  begin getV := v;
  end;

begin
end.
```

Including a NODEFAULT clause on an automated property is not allowed.

```
program Solve;

  type
     Base = class
       v : integer;
       procedure setV(x : integer);
       function getV : integer;
       automated
       property Value : integer read getV write setV;
     end;

  procedure Base.setV(x : integer);
  begin v := x;
  end;
```

```
    function Base.getV : integer;
    begin getV := v;
    end;

  begin
  end.
```

Removing the offending clause will cause the error to go away. Alternatively, moving the property out of the automated section will also make the error go away.

3.1.2.1.226 E2231: '%s' directive not allowed in dispinterface type

You have specified a clause in a dispinterface type which is not allowed.

```
program Produce;

  type
    IBase = dispinterface
    ['{00000000-0000-0000-0000-000000000000}']
      function Get : Integer;

      property BaseValue : Integer read Get;
    end;

    IExt = interface (IBase)
    end;


begin
end.
    program Solve;

  type
    IBase = dispinterface
    ['{00000000-0000-0000-0000-000000000000}']
      function Get : Integer;

      property BaseValue : Integer;
    end;

begin
end.
```

3.1.2.1.227 E2207: '%s' clause not allowed in interface type

The clause noted in the message is not allowed in an interface type. Typically this error indicates that an illegal directive has been specified for a property field in the interface.

```
program Produce;
  type
    Base = interface
      function Reader : Integer;
      procedure Writer(a : Integer);
      property Value : Integer read Reader write Writer stored false;
    end;
begin
end.
```

The problem in the above program is that the stored directive is not allowed in interface types.

```
program Solve;
  type
    Base = interface
```

```
      function Reader : Integer;
      procedure Writer(a : Integer);
      property Value : Integer read Reader write Writer;
    end;

begin
end.
```

The solution to problems of this nature are to remove the offending directive. Of course, it is best to understand the desired behavior and to implement it in some other fashion.

3.1.2.1.228 E2176: Illegal type in OLE automation section: '%s'

<typename> is not an allowed type in an OLE automation section. Only a small subset of all the valid Delphi language types are allowed in automation sections.

```
program Produce;

  type
    Base = class
      function GetC : Char;
      procedure SetC(c : Char);
    automated
      property Ch : Char read GetC write SetC dispid 151;
    end;

  procedure Base.SetC(c : Char);
  begin
  end;

  function Base.GetC : Char;
  begin GetC := '!';
  end;

begin
end.
```

Since the character type is not one allowed in the 'automated' section, the declaration of 'Ch' will produce an error when compiled.

```
program Solve;

  type
    Base = class
      function GetC : String;
      procedure SetC(c : String);
    automated
      property Ch : String read GetC write SetC dispid 151;
    end;

  procedure Base.SetC(c : String);
  begin
  end;

  function Base.GetC : String;
  begin GetC := '!';
  end;

begin
end.
```

There are two solutions to this problem. The first is to move the offending declaration out of the 'automated' section. The second is to change the offending type to one that is allowed in 'automated' sections.

3.1.2.1.229 E2185: Overriding automated virtual method '%s' cannot specify a dispid

The dispid declared for the original virtual automated procedure declaration must be used by all overriding procedures in derived classes.

```
program Produce;

  type
    Base = class
    automated
      procedure Automatic; virtual; dispid 151;
    end;


    Derived = class (Base)
    automated
      procedure Automatic; override; dispid 152;
    end;

  procedure Base.Automatic;
  begin
  end;

  procedure Derived.Automatic;
  begin
  end;

begin
end.
```

The overriding declaration of Base.Automatic, in Derived (Derived.Automatic) erroneously attempts to define another dispid for the procedure.

```
program Solve;

  type
    Base = class
    automated
      procedure Automatic; virtual; dispid 151;
    end;


    Derived = class (Base)
    automated
      procedure Automatic; override;
    end;

  procedure Base.Automatic;
  begin
  end;

  procedure Derived.Automatic;
  begin
  end;

begin
end.
```

By removing the offending dispid clause, the program will now compile.

3.1.2.1.230 E2068: Illegal reference to symbol '%s' in object file '%s'

This error message is given if an object file loaded with a $L or $LINK directive contains a reference to a Delphi symbol that is not a procedure, function, variable, typed constant or thread local variable.

3.1.2.1.231 E2139: Illegal message method index

You have specified value for your message index which <= 0.

```
program Produce;

  type
    Base = class
      procedure Dynamo(VAR x : Integer); message -151;
    end;

  procedure Base.Dynamo(VAR x : Integer);
  begin
  end;

begin
end.
```

The specification of -151 as the message index is illegal in the above example.

```
program Solve;

  type
    Base = class
      procedure Dynamo(VAR x : Integer); message 151;
    end;

  procedure Base.Dynamo(VAR x : Integer);
  begin
  end;

begin
end.
```

Always make sure that your message index values are >= 1.

3.1.2.1.232 E2224: $DESIGNONLY and $RUNONLY only allowed in package unit

The compiler has encountered either $designonly or $runonly in a source file which is not a package. These directives affect the way that the IDE will treat a package file, and therefore can only be contained in package source files.

3.1.2.1.233 E2184: %s section valid only in class types

Interfaces and records may not contain published sections.

Records may not contain protected sections.

3.1.2.1.234 W1043: Imagebase $%X is not a multiple of 64k. Rounding down to $%X

You can set an imagebase for a DLL to position it in a specific location in memory using the **$IMAGEBASE** compiler directive. The **$IMAGEBASE** directive controls the default load address for an application, DLL, or package. The number specified as the imagebase in the directive must be a multiple of 64K (that is, a hex number must have zeros as the last 4 digits), otherwise, it will be rounded down to the nearest multiple, and you will receive this compiler message.

3.1.2.1.235 E2227: Imagebase is too high - program exceeds 2 GB limit

There are three ways to cause this error: 1. Specify a large enough imagebase that, when compiled, the application code passes the 2GB boundary. 2. Specify an imagebase via the command line which is above 2GB. 3. Specify an imagebase via $imagebase which is above 2GB.

The only solution to this problem is to lower the imagebase address sufficiently so that the entire application will fit below the 2GB limit.

3.1.2.1.236 E2260: Implements clause not allowed together with index clause

You have tried to use an index clause with an implements clause. Index specifiers allow several properties to share the same access method while representing different values. The implements directive allows you to delegate implementation of an interface to a property in the implementing class but it cannot take an index specifier.

3.1.2.1.237 E2263: Implements getter cannot be dynamic or message method

An attempt has been made to use a dynamic or message method as a property accessor of a property which has an implements clause.

```
program Produce;
type
  I0 = interface
  end;

  T0 = class(TInterfacedObject, I0)
    function getter : I0; dynamic;
    property p0 : I0 read getter implements I0;
  end;

function T0.getter : I0;
begin
end;

end.
```

As shown in the example here, it is an error to use the dynamic modifier on a getter for a property which has an implements clause.

```
program Produce;
type
  I0 = interface
  end;

  T0 = class(TInterfacedObject, I0)
    function getter : I0;
    property p0 : I0 read getter implements I0;
  end;
```

```
function T0.getter : I0;
begin
end;

end.
```

To remove this error from your programs, remove the offending dynamic or method declaration.

3.1.2.1.238 E2264: Cannot have method resolutions for interface '%s'

An attempt has been made to use a method resolution clause for an interface named in an implements clause.

```
program Produce;
type
  I0 = interface
    procedure i0p0(a : char);
  end;

  T0 = class(TInterfacedObject, I0)
    procedure I0.i0p0 = proc0;
    function getter : I0;
    procedure proc0(a : char);
    property p0 : I0 read getter implements I0;
  end;

procedure T0.proc0(a : char);
begin
end;

function T0.getter : I0;
begin
end;
end.
```

In this example, the method proc0 is mapped onto the interface procedure i0p0, but because the interface is mentioned in a implements clause, this renaming is not allowed.

```
program Solve;
type
  I0 = interface
    procedure i0p0(a : char);
  end;

  T0 = class(TInterfacedObject, I0)
    function getter : I0;
    procedure i0p0(a : char);
    property p0 : I0 read getter implements I0;
  end;

procedure T0.i0p0(a : char);
begin
end;

function T0.getter : I0;
begin
end;
end.
```

The solution for this error is to remove the offending "name resolution clause". One easy way to accomplish this is to name the procedure in the class to the same name as the interface method.

3.1.2.1.239 E2258: Implements clause only allowed within class types

The interface definition in this example attempts to use an implements clause which causes the error.

```
program Produce;
type
  IMyInterface = interface
    function getter : IMyInterface;
    property MyInterface: IMyInterface read getter implements IMyInterface;
  end;
end.
```

The only viable solution to this problem is to remove the offending implements clause.

```
program Solve;
type
  IMyInterface = interface
    function getter : IMyInterface;
    property MyInterface: IMyInterface read getter;
  end;
end.
```

3.1.2.1.240 E2259: Implements clause only allowed for properties of class or interface type

An attempt has been made to use the implements clause with an improper type. Only class or interface types may be used.

```
program Produce;
type
  TMyClass = class(TInterfacedObject)
    FInteger : Integer;
    property MyInterface: Integer read FInteger implements Integer;
  end;
end.
```

In this example the error is caused because an Integer type is used with an implements clause.

The only solution for this error is to correct the implements clause so that it refers to a class or interface type, or to remove the offending clause altogether.

3.1.2.1.241 E2262: Implements getter must be %s calling convention

The compiler has encountered a getter or setter which does not have the correct calling convention.

```
program Produce;
type
  I0 = interface
  end;

  T0 = class(TInterfacedObject, I0)
    function getter : I0; cdecl;
    property p0 : I0 read getter implements I0;
  end;

function T0.getter : I0;
begin
end;
end.
```

As you can see in this example, the cdecl on the function getter causes this error to be produced.

```
program Solve;
```

```
type
  I0 = interface
  end;

  T0 = class(TInterfacedObject, I0)
    function getter : I0;
    property p0 : I0 read getter implements I0;
  end;

function T0.getter : I0;
begin
end;
end.
```

The only solution to this problem is to remove the offending calling convention from the property getter declaration.

3.1.2.1.242 E2265: Interface '%s' not mentioned in interface list

An implements clause references an interface which is not mentioned in the interface list of the class.

```
program Produce;
type
  IMyInterface = interface
  end;

  TMyClass = class(TInterfacedObject, IUnknown)
    FMyInterface: IMyInterface;
    property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
  end;
end.
```

The example shown here uses implements with the IMyInterface interface, but it is not mentioned in the interface list.

```
program Solve;
type
  IMyInterface = interface
  end;

  TMyClass = class(TInterfacedObject, IUnknown, IMyInterface)
    FMyInterface: IMyInterface;
    property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
  end;
end.
```

A quick solution, shown here, is to add the required interface to the interface list of the class definition. Of course, adding it to the interface list might require the implementation of the methods of the interface.

3.1.2.1.243 E2261: Implements clause only allowed for readable property

The compiler has encountered a "write only" property that claims to implement an interface. A property must be read/write to use the implements clause.

```
program Produce;
type
  IMyInterface = interface
  end;

  TMyClass = class(TInterfacedObject, IMyInterface)
    FMyInterface: IMyInterface;
    property MyInterface: IMyInterface implements IMyInterface;
  end;
end.
```

The property in this example is write only and cannot be used to implement an interface.

```
program Solve;
type
  IMyInterface = interface
  end;

  TMyClass = class(TInterfacedObject, IMyInterface)
    FMyInterface: IMyInterface;
    property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
  end;
end.
```

By adding a read clause, the property can use the implements clause.

3.1.2.1.244 x1033: Unit '%s' implicitly imported into package '%s'

The unit specified was not named in the contains clause of the package, but a unit which has already been included in the package imports it.

This message will help the programmer avoid violating the rule that a unit may not reside in more than one related package.

Ignoring the warning, will cause the unit to be put into the package. You could also explicitly list the named unit in the contains clause of the package to accomplish the same result and avoid the warning altogether. Or, you could alter the package list to load the named unit from another package.

```
package Produce;
  contains Classes;
end.
```

In the above program, Classes uses (either directly or indirectly) 'consts', 'TypInfo', and 'SysUtils'. We will get a warning message for each of these units.

```
package Solve;
  contains consts, TypInfo, SysUtils, Classes;
end.
```

The best solution for this problem is to explicitly name all the units which will be imported into the package in the contains clause, as has been done here.

3.1.2.1.245 W1040: Implicit use of Variants unit

If your application is using a Variant type, the compiler includes the Variant unit in the uses clause but warns you that you should add it explicitly.

3.1.2.1.246 E2420: Interface '%s' used in '%s' is not yet completely defined

Interface used in is not yet completely defined. Forward declared interfaces must be declared in the same type section that they are used in. As an example the following code will not compile because of the above error message:

```
program Project3;

{$APPTYPE CONSOLE}

type
  TInterface = interface;

  TFoo = class
    type
      TBar = class(TObject, TInterface)
        procedure Bar;
      end;
  end;

  TInterface = interface
    procedure Intf;
  end;

procedure TFoo.TBar.Bar;
begin

end;

begin
end.
```

3.1.2.1.247 E2086: Type '%s' is not yet completely defined

This error occurs if there is either a reference to a type that is just being defined, or if there is a forward declared class type in a type section and no final declaration of that type.

```
program Produce;

type
  TListEntry = record
    Next: ^TListEntry;                     (*<-- Error message here*)
    Data: Integer;
  end;
  TMyClass = class;                        (*<-- Error message here*)
  TMyClassRef = class of TMyClass;
  TMyClasss = class                        (*<-- Typo ...*)
    (*...*)
  end;

begin
end.
```

The example tries to refer to record type before it is completely defined. Also, because of a typo, the compiler never sees a complete declaration for TMyClass.

```
program Solve;

type
  PListEntry = ^TListEntry;
  TListEntry = record
    Next: PListEntry;
    Data: Integer;
  end;
  TMyClass = class;
  TMyClassRef = class of TMyClass;
  TMyClass = class
    (*...*)
  end;

begin
end.
```

The solution for the first problem is to introduce a type declaration for an auxiliary pointer type. The second problem is fixed by spelling TMyClass correctly.

3.1.2.1.248 E2195: Cannot initialize local variables

The compiler disallows the use of initialized local variables.

```
program Produce;
```

```
  var
    j : Integer;

  procedure Show;
    var i : Integer = 151;
  begin
  end;

begin
end.
```

The declaration and initialization of 'i' in procedure 'Show' is illegal.

```
program Solve;

  var
    j : Integer;

  procedure Show;
    var i : Integer;
  begin
    i := 151;
  end;

begin
  j := 0;
end.
```

You can use a programmatic style to set all variables to known values.

3.1.2.1.249 E2196: Cannot initialize multiple variables

Variable initialization can only occur when variables are declared individually.

```
program Produce;

  var
    i, j : Integer = 151, 152;

begin
end.
```

The compiler will disallow the declaration and initialization of more than one variable at a time.

```
program Solve;

  var
    i : Integer = 151;
    j : Integer = 152;

begin
end.
```

Simple declare each variable by itself to allow initialization.

3.1.2.1.250 E2194: Cannot initialize thread local variables

The compiler does not allow initialization of thread local variables.

```
program Produce;

  threadvar
    tls : Integer = 151;
```

```
begin
end.
```

The declaration and initialization of 'tls' above is not allowed.

```
program Solve;

  threadvar
    tls : Integer;

begin tls := 151;
end.
```

You can declare thread local storage as normal, and then initialize it in the initialization section of your source file.

3.1.2.1.251 E2072: Number of elements (%d) differs from declaration (%d)

This error message appears when you declare a typed constant or initialized variable of array type, but do not supply the appropriate number of elements.

```
program Produce;

var
  A : array [1..10] of Integer = (1,2,3,4,5,6,7,8,9);

begin
end.
```

The example declares an array of 10 elements, but the initialization only supplies 9 elements.

```
program Solve;

var
  A : array [1..10] of Integer = (1,2,3,4,5,6,7,8,9,10);

begin
end.
```

We just had to supply the missing element to make the compiler happy. When initializing bigger arrays, it can be sometimes hard to see whether you have supplied the right number of elements. To help with that, you layout the source file in a way that makes counting easy (e.g. ten elements to a line), or you can put the index of an element in comments next to the element itself.

3.1.2.1.252 E2428: Field '%s' needs initialization - not allowed in CLS compliant value types

CLS-compliant value types cannot have fields that require initialization. See ECMA 335, Partition II, Section 12.

3.1.2.1.253 E2418: Type '%s' needs initialization - not allowed in variant record

Type needs initialization - not allowed in variant record. Variant records do not allow types that need initialization in their variant field list since each variant field references the same memory location. As an example, the following code will not compile because the array type needs to be initialized.

```
program Project3;

{$APPTYPE CONSOLE}

type
  TFoo = record
    case Boolean of
      True: (bar: Integer);
      False: (baz: array [0..2] of Integer);
  end;

end.
```

3.1.2.1.254 E2426: Inline function must not have asm block

Inline functions can not include an asm block. To avoid this error, remove the inline directive from your function or use Pascal code to express the statements in the asm block.

3.1.2.1.255 E2442: Inline directive not allowed in constructor or destructor

Remove the inline directive to prevent this error.

3.1.2.1.256 H2444: Inline function '%s' has not been expanded because accessing member '%s' is inaccessible

An inline function cannot be expanded when the inline function body refers to a restricted member that is not accessible where the function is called.

For example, if an inline function refers to a **strict private** field and this function is called from outside the class (e.g. from a global procedure), the field is not accessible at the call site and the inline function is not expanded.

3.1.2.1.257 E2425: Inline methods must not be virtual nor dynamic

In order for an inline method to be inserted inline at compile-time, the method must be bound at compile-time. Virtual and dynamic methods are not bound until run-time, so they cannot be inserted inline. Make sure your method is static if you wish it to be inline.

3.1.2.1.258 E2449: Inlined nested routine '%s' cannot access outer scope variable '%s'

You can use the **inline** directive with nested procedures and functions. However, a nested procedure or function that refers to a variable that is local to the outer procedure is not eligible for inlining.

3.1.2.1.259 H2445: Inline function '%s' has not been expanded because its unit '%s' is specified in USES statement of IMPLEMENTATION section and current function is inline function or being inline function

Inline functions are not expanded between circularly dependent units.

3.1.2.1.260 H2443: Inline function '%s' has not been expanded because unit '%s' is not specified in USES list

This situation may occur if an inline function refers to a type in a unit that is not explicitly used by the function's unit. For example, this may happen if the function uses **inherited** to refer to methods inherited from a distant ancestor, and that ancestor's unit is not explicitly specified in the uses list of the function's unit.

If the inline function's code is to be expanded, then the unit that calls the function must explicitly use the unit where the ancestor type is exposed.

3.1.2.1.261 E2441: Inline function declared in interface section must not use local symbol '%s'

This error occurs when an inline function is declared in the interface section and it refers to a symbol that is not visible outside the unit. Expanding the inline function in another unit would require accessing the local symbol from outside the unit, which is not permitted.

To correct this error, move the local symbol declaration to the interface section, or make it an instance variable or class variable of the function's class type.

3.1.2.1.262 E2382: Cannot call constructors using instance variables

No further information is available for this error or warning.

3.1.2.1.263 E2102: Integer constant too large

You have specified an integer constant that requires more than 64 bits to represent.

```
program Produce;

  const
    VeryBigHex = $80000000000000001;

begin
end.
```

The constant in the above example is too large to represent in 64 bits, thus the compiler will output an error.

```
program Solve;

  const
    BigHex = $8000000000000001;

begin
end.
```

Check the constants that you have specified and ensure that they are representable in 64 bits.

3.1.2.1.264 F2084: Internal Error: %s%d

Occasionally when compiling an application in Delphi, the compile will halt and display an error message that reads, for example:

```
Internal Error: X1234
```

This error message indicates that the compiler has encountered a condition, other than a syntax error, that it cannot successfully process.

The information after "Internal Error" contains one or more characters, immediately followed by a number that indicates the file and line number in the compiler itself where the error occurred. Although this information may not help you, it can help us (Borland) track down the problem if and when you report the error. Be sure to jot down this information and include it with your internal error description.

See Also

Resolving internal errors (see page 130)

3.1.2.1.265 E2232: Interface '%s' has no interface identification

You have attempted to assign an interface to a GUID type, but the interface was not defined with a GUID.

```
program Produce;

  type
    IBase = interface
    end;

  var
    g : TGUID;

  procedure p(x : TGUID);
  begin
  end;

begin
  g := IBase;
  p(IBase);
end.
```

In this example, the IBase type is defined but it is not given an interface, and is thus cannot be assigned to a GUID type.

```
program Solve;

  type
    IBase = interface
    ['{00000000-0000-0000-0000-000000000000}']
    end;

  var
    g : TGUID;

  procedure p(x : TGUID);
  begin
  end;

begin
  g := IBase;
  p(IBase);
end.
```

To solve the problem, you must either not attempt to assign an interface type without a GUID to a GUID type, or you must assign a GUID to the interface when it is defined. In this solution, a GUID has been assigned to the interface type when it is defined.

3.1.2.1.266 E2291: Missing implementation of interface method %s.%s

This indicates that you have forgotten to implement a method required by an interface supported by your class type.

3.1.2.1.267 E2211: Declaration of '%s' differs from declaration in interface '%s'

A method declared in a class which implements an interface is different from the definition which appears in the interface. Probable causes are that a parameter type or return value is declared differently, the method appearing in the class is a message method, the identifier in the class is a field or the identifier in the class is a property, which does not match with the definition in the interface.

```
program Produce;

  type
    IBaseIntf = interface
```

```
        procedure p0(var x : Shortint);
        procedure p1(var x : Integer);
        procedure p2(var x : Integer);
      end;

      TBaseClass = class (TInterfacedObject)
        procedure p1(var x : Integer); message 151;
      end;

      TExtClass = class (TBaseClass, IBaseIntf)
        p2 : Integer;
        procedure p0(var x : Integer);
        procedure p1(var x : Integer); override;
      end;

    procedure TBaseClass.p1(var x : Integer);
    begin
    end;

    procedure TExtClass.p0(var x : Integer);
    begin
    end;

    procedure TExtClass.p1(var x : Integer);
    begin
    end;

  begin
  end.
```

Generally, as in this example, errors of this type are plain enough to be easily visible. However, as can be seen with p1, things can be more subtle. Since p1 is overriding a procedure from the inherited class, p1 also inherits the virtuality of the procedure defined in the base class.

```
program Solve;

  type
    IBaseIntf = interface
      procedure p0(var x : Shortint);
      procedure p1(var x : Integer);
      procedure p2(var x : Integer);
    end;

    TBaseClass = class (TInterfacedObject)
      procedure p1(var x : Integer); message 151;
    end;

    TExtClass = class (TBaseClass, IBaseIntf)
      p2 : Integer;

      procedure IBaseIntf.p1 = p3;
      procedure IBaseIntf.p2 = p4;

      procedure p0(var x : Shortint);
      procedure p1(var x : Integer); override;
      procedure p3(var x : Integer);
      procedure p4(var x : Integer);
    end;

  procedure TBaseClass.p1(var x : Integer);
  begin
  end;

  procedure TExtClass.p0(var x : Shortint);
  begin
  end;
  procedure TExtClass.p1(var x : Integer);
  begin
  end;

  procedure TExtClass.p3(var x : Integer);
  begin
  end;

  procedure TExtClass.p4(var x : Integer);
  begin
  end;

begin
end.
```

One approach to solving this problem is to use a message resolution clause for each problematic identifier, as is done in the example shown here. Another viable approach, which requires more thoughtful design, would be to ensure that the class identifiers are compatible to the interface identifiers before compilation.

3.1.2.1.268 E2208: Interface '%s' already implemented by '%s'

The class specified by name2 has specified the interface name1 more than once in the inheritance section of the class definition.

```
program Produce;
  type
    IBaseIntf = interface
    end;

    TBaseClass = class (TInterfacedObject, IBaseIntf, IBaseIntf)
    end;

begin
end.
```

In this example, the IBaseIntf interface is specified multiple times in the inheritance section of the definition of TBaseClass. As a class can not implement the same interface more than once, this cause the compiler to emit the error message.

```
program Solve;

  type
    IBaseIntf = interface
    end;

    TBaseClass = class (TInterfacedObject, IBaseIntf)
    end;

begin
end.
```

The only solution to this error message is to ensure that a particular interface appears no more than once in the inheritance section of a class definition.

3.1.2.1.269 E2089: Invalid typecast

This error message is issued for type casts not allowed by the rules. The following kinds of casts are allowed:

- Ordinal or pointer type to another ordinal or pointer type
- A character, string, array of character or pchar to a string
- An ordinal, real, string or variant to a variant
- A variant to an ordinal, real, string or variant

- A variable reference to any type of the same size.

Note that casting real types to integer can be performed with the standard functions Trunc and Round.

There are other transfer functions like Ord and Chr that might make your intention clearer.

```
program Produce;

begin
  Writeln( Integer(Pi) );
end.
```

This programmer thought he could cast a floating point constant to Integer, like in C.

```
program Solve;

begin
  Writeln( Trunc(Pi) );
end.
```

In the Delphi language, we have separate Transfer functions to convert floating point values to integer.

3.1.2.1.270 E2424: Codepage '%s' is not installed on this machine

This message occurs if you specify a codepage using the `--codepage=nnn` command line switch and the codepage you specify is not available on the machine.

See your operating system documentation for details on how to install codepages.

3.1.2.1.271 E2173: Missing or invalid conditional symbol in '$%s' directive

The $IFDEF, $IFNDEF, $DEFINE and $UNDEF directives require that a symbol follow them.

```
program Produce;

(*$IFDEF*)
(*$ENDIF*)

begin
end.
```

The $IFDEF conditional directive is incorrectly specified here and will result in an error.

```
program Solve;

(*$IFDEF WIN32*)
(*$ENDIF*)

begin
end.
```

The solution to the problem is to ensure that a symbol to test follows the appropriate directives.

3.1.2.1.272 x1030: Invalid compiler directive: '%s'

This error message means there is an error in a compiler directive or in a command line option. Here are some possible error situations:

- An external declaration was syntactically incorrect.
- A command line option or an option in a DCC32.CFG file was not recognized by the compiler or was invalid. For example, '-$M100' is invalid because the minimum stack size must be at least 1024.
- The compiler found a $XXXXX directive, but could not recognize it. It was probably misspelled.

- The compiler found a $ELSE or $ENDIF directive, but no preceding $IFDEF, $IFNDEF or $IFOPT directive.
- (*$IFOPT*) was not followed by a switch option and a + or -.
- The long form of a switch directive was not followed by ON or OFF.
- A directive taking a numeric parameter was not followed by a valid number.
- The $DESCRIPTION directive was not followed by a string.
- The $APPTYPE directive was not followed by CONSOLE or GUI.
- The $ENUMSIZE directive (short form $Z) was not followed by 1,2 or 4.

```
(*$Description Copyright CodeGear 2007*)    (*<-- Error here*)
program Produce;
(*$AppType Console*)                                     (*<-- Error here*)

begin
(*$If O+*)                                               (*<-- Error here*)
Writeln('Optimizations are ON');
(*$Else*)                                                (*<-- Error here*)
Writeln('Optimizations are OFF');
(*$Endif*)                                               (*<-- Error here*)
Writeln('Hello world!');
end.
```

The example shows three typical error situations, and the last two errors are caused by the compiler not having recognized $If.

```
(*$Description 'Copyright CodeGear 2007'*)  (*Need string*)
program Solve;
(*$AppType Console*)                                     (*AppType*)

begin
(*$IfOpt O+*)                                            (*IfOpt*)
  Writeln('Optimizations are ON');
(*$Else*)                                                (*Now fine*)
  Writeln('Optimizations are OFF');
(*$Endif*)                                               (*Now fine*)
  Writeln('Hello world!');
end.
```

So $Description needs a quoted string, we need to spell $AppType right, and checking options is done with $IfOpt. With these changes, the example compiles fine.

3.1.2.1.273 E2298: read/write not allowed for CLR events. Use Include/Exclude procedure

Multicast events cannot be assigned to or read from like traditional Delphi read/write events.

Use Include/Exclude to add or remove methods.

3.1.2.1.274 E2138: Invalid message parameter list

A message procedure can take only one, VAR, parameter; it's type is not checked.

```
program Produce;

  type
    Base = class
      procedure Msg1(x : Integer); message 151;
      procedure Msg2(VAR x, y : Integer); message 152;
    end;
  procedure Base.Msg1(x : Integer);
  begin
  end;

  procedure Base.Msg2(VAR x, y : Integer);
  begin
  end;

begin
end.
```

The obvious error in the first case is that the parameter is not VAR. The error in the second case is that more than one parameter is declared.

```
program Solve;

  type
    Base = class
      procedure Msg1(VAR x : Integer); message 151;
      procedure Msg2(VAR y : Integer); message 152;
    end;

  procedure Base.Msg1(VAR x : Integer);
  begin
  end;

  procedure Base.Msg2(VAR y : Integer);
  begin
  end;

begin
end.
```

The solution in both cases was to only specify one, VAR, parameter in the message method declaration.

3.1.2.1.275 E2294: A class helper that descends from '%s' can only help classes that are descendents '%s'

The object type specified in the "for" clause of a class helper declaration is not a descendent of the object type specified in the "for" clause of the class helper's ancestor type.

3.1.2.1.276 E2296: A constructor introduced in a class helper must call the parameterless constructor of the helped class as the first statement

The first statement in a class helper constructor must be "inherited Create;"

3.1.2.1.277 E2387: The key container name '%s' does not exist

No further information is available for this error or warning.

3.1.2.1.278 E2388: Unrecognized strong name key file '%s'

No further information is available for this error or warning.

3.1.2.1.279 E2432: %s cannot be applied to a rectangular dynamic array

This error may arise if you attempt to pass a dynamically allocated rectangular array to the Low or High function.

If you receive this error, use a static or ragged (non-rectangular) array. See the Delphi Language Guide for details.

See Also

Structured Types (↗ see page 566)

3.1.2.1.280 E2393: Invalid operator declaration

No further information is available for this error or warning.

3.1.2.1.281 E2174: '%s' not previously declared as a PROPERTY

You have attempted to hoist a property to a different visibility level by redeclaration, but <name> in the base class was not declared as a property. -W

```
program Produce;
(*$WARNINGS ON*)

  type
    Base = class
    protected
      Caption : String;
      Title : String;
      property TitleProp : string read Title write Title;
    end;

    Derived = class (Base)
    public
      property Title read Caption write Caption;
    end;

begin
end.
```

The intent of the redeclaration of 'Derived.Title' is to change the field which is used to read and write the property 'Title' as well as hoist it to 'public' visibility. Unfortunately, the programmer really meant to use 'TitleProp', not 'Title'.

```
program Solve;
(*$WARNINGS ON*)

  type
    Base = class
    protected
      Caption : String;
      Title : String;
      property TitleProp : string read Title write Title;
    end;

    Derived = class (Base)
    public
      property TitleProp read Caption write Caption;
      property Title : string read Caption write Caption;
    end;

begin
end.
```

There are a couple ways of approaching this error. The first, and probably the most commonly taken, is to specify the real property which is to be redeclared. The second, which can be seen in the redeclaration of 'Title' addresses the problem by explicitly creating a new property, with the same name as a field in the base class. This new property will hide the base field, which will no longer be accessible without a typecast. (Note: If you have warnings turned on, the redeclaration of 'Title' will issue a warning notifying you that the redeclaration will hide the base class' member.)

3.1.2.1.282 E2376: STATIC can only be used on non-virtual class methods

No further information is available for this error or warning.

3.1.2.1.283 E2415: Could not import assembly '%s' because it contains namespace '%s'

The Borland.Delphi.System unit may only be loaded from the Borland.Delphi.dll assembly. This error will occur if Borland.Delphi.System is attempted to be loaded from an alternative assembly.

3.1.2.1.284 E2416: Could not import package '%s' because it contains system unit '%s'

The Borland.Delphi.System unit may only be loaded from the Borland.Delphi.dll package. This error will occur if Borland.Delphi.System is attempted to be loaded from an alternative package.

3.1.2.1.285 F2438: UCS-4 text encoding not supported. Convert to UCS-2 or UTF-8

This error is encountered when a source file has a UCS-4 encoding, as indicated by its Byte-Order-Mark (BOM). The compiler does not support compilation of source files in UCS-4 Unicode encoding. To solve this problem, convert the source file to UCS-2 or UTF-8 encoding.

3.1.2.1.286 E2386: Invalid version string '%s' specified in %s

No further information is available for this error or warning.

3.1.2.1.287 E2120: LOOP/JCXZ distance out of range

You have specified a LOOP or JCXZ destination which is out of range. You should not receive this error as the jump range is 2Gb for LOOP and JCXZ instructions.

3.1.2.1.288 E2049: Label declaration not allowed in interface part

This error occurs when you declare a label in the interface part of a unit.

```
unit Produce;
interface
label 99;
implementation
begin
99:
end.
```

It is just illegal to declare a label in the interface section of a unit.

```
unit Solve;
interface
implementation
label 99;
begin
99:
end.
```

You have to move it to the implementation section.

3.1.2.1.289 E2073: Label already defined: '%s'

This error message is given when a label is set on more than one statement.

```
program Produce;
label 1;
begin
1:
  goto 1;
1:        (*<-- Error message here*)
end.
```

The example just tries to set label 1 twice.

```
program Solve;
label 1;
begin
1:
  goto 1;
end.
```

Make sure every label is set exactly once.

3.1.2.1.290 E2074: Label declared and referenced, but not set: '%s'

You declared and used a label in your program, but the label definition was not encountered in the source code.

```
  program Produce;

    procedure Labeled;
    label 10;
    begin
      goto 10;
    end;

  begin
  end.
```

Label 10 is declared and used in the procedure 'Labeled', but the compiler never finds a definition of the label.

```
  program Produce;

    procedure Labeled;
    label 10;
    begin
      goto 10;
      10:
    end;

  begin
  end.
```

The simple solution is to ensure that a declared and used label has a definition, in the same scope, in your program.

3.1.2.1.291 F2069: Line too long (more than 1023 characters)

This error message is given when the length of a line in the source file exceeds 255 characters.

Usually, you can divide the long line into two shorter lines.

If you need a really long string constant, you can break it into several pieces on consecutive lines that you concatenate with the '+' operator.

3.1.2.1.292 E2364: Cross-assembly protected reference to [%s]%s.%s in %s.%s

In Delphi for .NET, members with protected visibility cannot be accessed outside of the assembly in which they are defined. If possible, you may want to use the publicly-exposed members of the class to accomplish your goal.

Other ways to resolve this error:

- Increase the visibility of the member from protected to public, so it can be accessed outside of its assembly.
- “Link in” the assembly where the protected member is defined, so that this assembly is incorporated into the assembly you are building, and the access will be inside the assembly.

See Also

Classes and Objects (see page 514)

Linking Delphi Units into an Application (see page 111)

3.1.2.1.293 W1053: Local PInvoke code has not been made because external routine '%s' in package '%s' is defined using package local types in its custom attributes

This warning may arise when an external package uses PInvoke to access Win32 library code, and that package exposes the PInvoke definition through a public export. In these cases the compiler will attempt to link directly to the Win32 library by copying the PInvoke definition to the local assembly, rather than linking to the public export in the external package. This is more secure and can also improve runtime performance.

This warning message is issued if the compiler is unable to emit the PInvoke definition locally, because the external assembly uses locally-defined types for a custom attribute. To avoid this warning, you must change the named package so that it does not use locally-defined types for a custom attribute in an exported function or procedure.

For example, in the following code, the unit `ExternalPackagedUnit` exposes the external function `Beep` in `kernel32` through the `TFoo.Beep` function, with the `MyAttribute` custom attribute:

```
unit ExternalPackagedUnit;

interface

function Beep(dwFreq, dwDuration: MyLongWord): Boolean; static; stdcall;

implementation

type
  MyAttribute = class(System.Attribute)
  .
  .
  .
  end;
```

```
[MyAttribute]
function Beep(dwFreq, dwDuration: LongWord): Boolean; stdcall; external 'kernel32' name 'Beep';

end.
```

If one attempts to compile a program which uses ExternalPackagedUnit, then because the MyAttribute type is locally-defined in ExternalPackagedUnit the compiler will be unable to link directly to the Beep function in kernel32, and this warning will be issued. In the example, this problem can be solved by making the MyAttribute type public (by moving its declaration from the **implementation** section to the **interface** section), or by removing the custom attribute from the Beep function.

See Also

Using Platform Invoke with RAD Studio

3.1.2.1.294 E2094: Local procedure/function '%s' assigned to procedure variable

This error message is issued if you try to assign a local procedure to a procedure variable, or pass it as a procedural parameter.

This is illegal, because the local procedure could then be called even if the enclosing procedure is not active. This situation would cause the program to crash if the local procedure tried to access any variables of the enclosing procedure.

```
program Produce;

var
  P: Procedure;

procedure Outer;

  procedure Local;
  begin
    Writeln('Local is executing');
  end;

begin
  P := Local;       (*<-- Error message here*)
end;

begin
  Outer;
  P;
end.
```

The example tries to assign a local procedure to a procedure variable. This is illegal because it is unsafe at runtime.

```
program Solve;

var
  P: Procedure;

procedure NonLocal;
begin
  Writeln('NonLocal is executing');
end;

procedure Outer;

begin
  P := NonLocal;
end;

begin
```

```
  Outer;
  P;
end.
```

The solution is to move the local procedure out of the enclosing one.

3.1.2.1.295 E2189: Thread local variables cannot be local to a function

Thread local variables must be declared at a global scope.

```
program Produce;

  procedure NoTLS;
    threadvar
      x : Integer;
  begin
  end;

begin
end.
```

A thread variable cannot be declared local to a procedure.

```
program Solve;

  threadvar
    x : Integer;

  procedure YesTLS;
    var
      localX : Integer;
  begin
  end;

begin
end.
```

There are two simple alternatives for avoiding this error. First, the threadvar section can be moved to a local scope. Secondly, the threadvar in the procedure could be changed into a normal var section. Note that if compiler hints are turned on, a hint about localX being declared but not used will be emitted.

3.1.2.1.296 W1042: Error converting locale string '%s' to Unicode. String truncated. Is your LANG environment variable set correctly?

This message occurs when you are trying to convert strings to Unicode and the string contains characters that are not valid for the current locale. For example, this may occur when converting WideString to AnsiString or if attempting to display Japanese characters in an English locale.

3.1.2.1.297 E2011: Low bound exceeds high bound

This error message is given when either the low bound of a subrange type is greater than the high bound, or the low bound of a case label range is greater than the high bound.

```
program Produce;
type
  SubrangeType = 1..0;              (*Gets: Low bound exceeds high bound *)
begin
  case True of
  True..False:                      (*Gets: Low bound exceeds high bound *)
    Writeln('Expected result');
  else
    Writeln('Unexpected result');
  end;
end.
```

In the example above, the compiler gives an error rather than treating the ranges as empty. Most likely, the reversal of the bounds was not intentional.

```
program Solve;
type
  SubrangeType = 0..1;
begin
  case True of
  False..True:
    Writeln('Expected result');
  else
    Writeln('Unexpected result');
  end;
end.
```

Make sure you have specified the bounds in the correct order.

3.1.2.1.298 H2440: Inline method visibility is not lower or same visibility of accessing member '%s.%s'

A member that is accessed within the body of an inline method must be accessible anywhere that the inline method is called. Therefore, the member must be at least as visible as the inline method.

Here is an example of code that will raise this error:

```
type
   TFoo = class
   private
     PrivateMember: Integer;
   public
     function PublicFunc:Integer; inline;
   end;

function TFoo.PublicFunc:Integer;
    begin
        Result := Self.PrivateMember;
    end;
```

Because `Result := Self.PrivateMember;` will be inserted wherever PublicFunc is called, PrivateMember must be accessible in any such location.

To correct this error, remove the inline directive or adjust the visibility of the inline method or the member it accesses.

3.1.2.1.299 E2204: Improper GUID syntax

The GUID encountered in the program source is malformed. A GUID must be of the form: 00000000-0000-0000-0000-000000000000.

3.1.2.1.300 E2348: Metadata - Bad input parameters

No further information is available for this error or warning.

3.1.2.1.301 E2347: Metadata - Bad binary signature

No further information is available for this error or warning.

3.1.2.1.302 E2349: Metadata - Cannot resolve typeref

No further information is available for this error or warning.

3.1.2.1.303 E2345: Metadata - Attempt to define an object that already exists

No further information is available for this error or warning.

3.1.2.1.304 E2346: Metadata - A guid was not provided where one was required

No further information is available for this error or warning.

3.1.2.1.305 E2350: Metadata - No logical space left to create more user strings

No further information is available for this error or warning.

3.1.2.1.306 F2046: Out of memory

The compiler ran out of memory.

This should rarely happen. If it does, make sure your swap file is large enough and that there is still room on the disk.

3.1.2.1.307 x1054: Linker error: %s

This message emits a warning or other text generated using the $MESSAGE directive.

3.1.2.1.308 E2096: Method identifier expected

This error message will be issued in several different situations:

- Properties in an automated section must use methods for access, they cannot use fields in their read or write clauses.
- You tried to call a class method with the "ClassType.MethodName" syntax, but "MethodName" was not the name of a method.
- You tried calling an inherited with the "Inherited MethodName" syntax, but "MethodName" was not the name of a method.

```
program Produce;

type
TMyBase = class
  Field: Integer;
end;
TMyDerived = class (TMyBase)
  Field: Integer;
  function Get: Integer;
Automated
  property Prop: Integer read Field;    (*<-- Error message here*)
end;
```

```
function TMyDerived.Get: Integer;
begin
Result := TMyBase.Field;                (*<-- Error message here*)
end;

begin
end.
```

The example tried to declare an automated property that accesses a field directly. The second error was caused by trying to get at a field of the base class - this is also not legal.

```
program Solve;

type
  TMyBase = class
    Field: Integer;
  end;
  TMyDerived = class (TMyBase)
    Field: Integer;
    function Get: Integer;
  Automated
    property Prop: Integer read Get;
  end;

function TMyDerived.Get: Integer;
begin
  Result := TMyBase(Self).Field;
end;

begin
  Writeln( TMyDerived.Create.Prop );
end.
```

The first problem is fixed by accessing the field via a method. The second problem can be fixed by casting the Self pointer to the base class type, and accessing the field off of that.

3.1.2.1.309 E2433: Method declarations not allowed in anonymous record or local record type

Record types that are declared in local scopes or declared in-place in variable declarations can only contain field declarations. For advanced features in record types (such as methods, properties, and nested types), the record type must be an explicitly declared global type.

3.1.2.1.310 E2234: Getter or setter for property '%s' cannot be found

During translation of a unit to a C++ header file, the compiler is unable to locate a named symbol which is to be used as a getter or setter for a property. This is usually caused by having nested records in the class and the accessor is a field in the nested record.

3.1.2.1.311 E2095: Missing ENDIF directive

This error message is issued if the compiler does not find a corresponding $ENDIF directive after an $IFDEF, $IFNDEF or $IFOPT directive.

```
program Produce;
(*$APPTYPE CONSOLE*)
begin
(*$IfOpt O+*)
  Writeln('Compiled with optimizations');
```

```
(*$Else*)
  Writeln('Compiled without optimizations');
(*Endif*)
end.                                           (*<-- Error message here*)
```

In this example, we left out the $ character in the (*$Endif*) directive, so the compiler mistook it for a comment.

```
program Solve;
(*$APPTYPE CONSOLE*)
begin
(*$IfOpt O+*)
  Writeln('Compiled with optimizations');
(*$Else*)
  Writeln('Compiled without optimizations');
(*$Endif*)
end.
```

The solution is to make sure all the conditional directives have a valid $ENDIF directive.

3.1.2.1.312 E2403: Add or remove accessor for event '%s' cannot be found

No further information is available for this error or warning.

3.1.2.1.313 E2253: Ancestor type '%s' does not have an accessible default constructor

The ancestor of the class being compiled does not have an accessible default constructor. This error only occurs with the byte code version of the compiler.

3.1.2.1.314 E2066: Missing operator or semicolon

This error message appears if there is no operator between two subexpressions, or no semicolon between two statements.

Often, a semicolon is missing on the previous line.

```
program Produce;
var
  I: Integer;
begin
  I := 1 2                   (*<-- Error message here*)
  if I = 3 then              (*<-- Error message here*)
  Writeln('Fine')
end.
```

The first statement in the example has two errors - a '+' operator and a semicolon are missing. The first error is reported on this statement, the second on the following line.

```
program Solve;
var
  I: Integer;
begin
  I := 1 + 2;                (*We were missing a '+' operator and a semicolon*)
  if I = 3 then
  Writeln('Fine')
end.
```

The solution is to make sure the necessary operators and semicolons are there.

3.1.2.1.315 E2202: Required package '%s' not found

The package, which is referenced in the message, appears on the package list, either explicitly or through a requires clause of another unit appearing on the package list, but cannot be found by the compiler.

The solution to this problem is to ensure that the DCP file for the named package is in one of the units named in the library path.

3.1.2.1.316 E2035: Not enough actual parameters

This error message occurs when a call to procedure or function gives less parameters than specified in the procedure or function declaration.

This can also occur for calls to standard procedures or functions.

```
program Produce;
var
  X: Real;
begin
  Val('3.141592', X);   (*<-- Error message here*)
end.
```

The standard procedure Val has one additional parameter to return an error code in. The example did not supply that parameter.

```
program Solve;
var
  X: Real;
  Code: Integer;
begin
  Val('3.141592', X, Code);
end.
```

Typically, you will check the call against the declaration of the procedure called or the help, and you will find you forgot about a parameter you need to supply.

3.1.2.1.317 E2067: Missing parameter type

This error message is issued when a parameter list gives no type for a value parameter.

Leaving off the type is legal for constant and variable parameters.

```
program Produce;

procedure P(I;J: Integer);                            (*<-- Error message here*)
begin
end;

function ComputeHash(Buffer; Size: Integer): Integer; (*<-- Error message here*)
begin
end;


begin
end.
```

We intended procedure P to have two integer parameters, but we put a semicolon instead of a comma after the first parameters. The function ComputeHash was supposed to have an untyped first parameter, but untyped parameters must be either variable or constant parameters - they cannot be value parameters.

```
program Solve;

procedure P(I,J: Integer);
```

```
begin
end;

function ComputeHash(const Buffer; Size: Integer): Integer;
begin
end;

begin
end.
```

The solution in this case was to fix the type in P's parameter list, and to declare the Buffer parameter to ComputeHash as a constant parameter, because we don't intend to modify it.

3.1.2.1.318 E2151: Could not load RLINK32.DLL

RLINK32 could not be found. Please ensure that it is on the path.

Contact CodeGear if you encounter this error.

3.1.2.1.319 E2404: Cannot mix READ/WRITE property accessors with ADD/REMOVE accessors

No further information is available for this error or warning.

3.1.2.1.320 E2359: Multiple class constructors in class %s: %s and %s

No further information is available for this error or warning.

3.1.2.1.321 E2287: Cannot export '%s' multiple times

This message is not used in this product.

3.1.2.1.322 E2085: Unit name mismatch: '%s' '%s'

The unit name in the top unit is case sensitive and must match the name with respect to upper- and lowercase letters exactly. The unit name is case sensitive only in the unit declaration.

3.1.2.1.323 E2016: Array type required

This error message is given if you either index into an operand that is not an array, or if you pass an argument that is not an array to an open array parameter.

```
program Produce;
var
  P: ^Integer;
  I: Integer;
begin
  Writeln(P[I]);
end.
```

We try to apply an index to a pointer to integer - that would be legal in C, but is not in Delphi.

```
program Solve;
type
  TIntArray = array [0..MaxInt DIV sizeof(Integer)-1] of Integer;
```

```
var
  P: ^TIntArray;
  I: Integer;
begin
  Writeln(P^[I]);   (*Actually, P[I] would also be legal*)
end.
```

In The Delphi language, we must tell the compiler that we intend P to point to an array of integers.

3.1.2.1.324 E2012: Type of expression must be BOOLEAN

This error message is output when an expression serves as a condition and must therefore be of Boolean type. This is the case for the controlling expression of the if, while and repeat statements, and for the expression that controls a conditional breakpoint.

```
program Produce;
var
  P: Pointer;
begin
  if P then
    Writeln('P <> nil');
end.
```

Here, a C++ programmer just used a pointer variable as the condition of an if statement.

```
program Solve;
var
  P: Pointer;
begin
  if P <> nil then
    Writeln('P <> nil');
end.
```

In Delphi, you need to be more explicit in this case.

3.1.2.1.325 E2021: Class type required

In certain situations the compiler requires a class type:

- As the ancestor of a class type
- In the on-clause of a try-except statement
- As the first argument of a raise statement
- As the final type of a forward declared class type

```
program Produce;
begin
raise 'This would work in C++, but does not in Delphi';
end.
  program Solve;
uses SysUtils;
begin
raise Exception.Create('There is a simple workaround, however');
end.
```

3.1.2.1.326 E2076: This form of method call only allowed for class methods

You were trying to call a normal method by just supplying the class type, not an actual instance.

This is only allowed for class methods and constructors, not normal methods and destructors.

```
program Produce;
```

```
type
  TMyClass = class
  (*...*)
  end;
var
  MyClass: TMyClass;

begin
  MyClass := TMyClass.Create;  (*Fine, constructor*)
  Writeln(TMyClass.ClassName); (*Fine, class method*)
  TMyClass.Destroy;            (*<-- Error message here*)
end.
```

The example tries to destroy the type TMyClass - this doesn't make sense and is therefore illegal.

```
program Solve;
type
  TMyClass = class
  (*...*)
  end;
var
  MyClass: TMyClass;

begin
  MyClass := TMyClass.Create;  (*Fine, constructor*)
  Writeln(TMyClass.ClassName); (*Fine, class method*)
  MyClass.Destroy;             (*Fine, called on instance*)
end.
```

As you can see, we really meant to destroy the instance of the type, not the type itself.

3.1.2.1.327 E2149: Class does not have a default property

You have used a class instance variable in an array expression, but the class type has not declared a default array property.

```
program Produce;

  type
    Base = class
    end;

  var
    b : Base;

  procedure P;
    var ch : Char;
  begin
    ch := b[1];
  end;

begin
end.
```

The example above elicits an error because 'Base' does not declare an array property, and 'b' is not an array itself.

```
program Solve;

  type
    Base = class
      function GetChar(i : Integer) : Char;
      property data[i : Integer] : Char read GetChar; default;
    end;

  var
    b : Base;
  function Base.GetChar(i : Integer) : Char;
  begin GetChar := 'A';
  end;

  procedure P;
    var ch : Char;
  begin
    ch := b[1];
    ch := b.data[1];
  end;

begin
end.
```

When you have declared a default property for a class, you can use the class instance variable in array expression, as if the class instance variable itself were actually an array. Alternatively, you can use the name of the property as the actual array accessor.

Note: If you have hints turned on, you will receive two warnings about the value assigned to 'ch' never being used.

3.1.2.1.328 E2168: Field or method identifier expected

You have specified an identifier for a read or write clause to a property which is not a field or method.

```
program Produce;

  var
    r : string;

  type
    Base = class
      t : string;
      property Title : string read Title write Title;
      property Caption : string read r write r;

    end;

begin
end.
```

The two properties in this code both cause errors. The first causes an error because it is not possible to specify the property itself as the read & write methods. The second causes an error because 'r' is not a member of the Base class.

```
program Solve;

  type
    Base = class
      t : string;
      property Title : string read t write t;
    end;

begin
end.
```

To solve this error, make sure that all read & write clauses for properties specify a valid field or method identifier that is a member of the class which owns the property.

3.1.2.1.329 E2022: Class helper type required

When declaring a class helper type with an ancestor clause, the ancestor type must be a class helper.

3.1.2.1.330 E2380: Instance or class static method expected

No further information is available for this error or warning.

3.1.2.1.331 E2013: Type of expression must be INTEGER

This error message is only given when the constant expression that specifies the number of characters in a string type is not of type integer.

```
program Produce;
type
  color = (red,green,blue);
var
  S3 : string[Succ(High(color))];
begin
end.
```

The example tries to specify the number of elements in a string as dependent on the maximum element of type color - unfortunately, the element count is of type color, which is illegal.

```
program Solve;
type
  color = (red,green,blue);
var
  S3 : string[ord(High(color))+1];
begin
end.
```

3.1.2.1.332 E2205: Interface type required

A type, which is an interface, was expected but not found. A common cause of this error is the specification of a user-defined type that has not been declared as an interface type.

```
program Produce;
  type
    Name = string;

    MyObject = class
    end;

    MyInterface = interface(MyObject)
    end;

    Base = class(TObject, Name)
    end;

begin
end.
```

In this example, the type 'Base' is erroneously declared since 'Name' is not declared as an interface type. Likewise, 'MyInterface' is incorrectly declared because its ancestor interface was not declared as such.

```
program Solve;
  type
    BaseInterface = interface
    end;

    MyInterface = interface(BaseInterface)
    end;

    Base = class(TObject, MyInterface)
    end;
```

```
begin
end.
```

The best solution when encountering this error is to reexamine the source code to determine what was really intended. If a class is to implement an interface, it must first be explicitly derived from a base type such as TObject. When extended, interfaces can only have a single interface as its ancestor.

In the example above, the interface is properly derived from another interface and the object definition correctly specifies a base so that interfaces can be specified.

3.1.2.1.333 E2031: Label expected

This error message occurs if the identifier given in a goto statement or used as a label in inline assembly is not declared as a label.

```
program Produce;

begin
  if 2*2 <> 4 then
    goto Exit; (*<-- Error message here: Exit is also a standard procedure*)
  (*...*)
Exit:              (*Additional error messages here*)
end.
    program Solve;
label
  Exit;            (*Labels must be declared in Delphi*)
begin
  if 2*2 <> 4 then
    goto Exit;
  (*...*)
Exit:
end.
```

3.1.2.1.334 E2075: This form of method call only allowed in methods of derived types

This error message is issued if you try to make a call to a method of an ancestor type, but you are in fact not in a method.

```
program Produce;

type
  TMyClass = class
    constructor Create;
  end;

procedure Create;
begin
  inherited Create;      (*<-- Error message here*)
end;

begin
end.
```

The example tries to call an inherited constructor in procedure Create, which is not a method.

```
program Solve;

type
  TMyClass = class
    constructor Create;
  end;
```

```
constructor TMyclass.Create;
begin
  inherited Create;
end;

begin
end.
```

The solution is to make sure you are in fact in a method when using this form of call.

3.1.2.1.335 E2019: Object type required

This error is given whenever an object type is expected by the compiler. For instance, the ancestor type of an object must also be an object type.

```
type
  MyObject = object(TObject)
  end;
begin
end.
```

Confusingly enough, TObject in the unit System has a class type, so we cannot derive an object type from it.

```
program Solve;
type
  MyObject = class  (*Actually, this means: class(TObject)*)
  end;
begin
end.
```

Make sure the type identifier really stands for an object type - maybe it is misspelled, or maybe is hidden by an identifier from another unit.

3.1.2.1.336 E2020: Object or class type required

This error message is given when the syntax 'Typename.Methodname' is used, but the typename does not refer to an object or class type.

```
program Produce;
type
  TInteger = class
    Value: Integer;
  end;
var
  V: TInteger;
begin
  V := Integer.Create;
end.
```

Type Integer does not have a Create method, but TInteger does.

```
program Solve;
type
  TInteger = class
    Value: Integer;
  end;
var
  V: TInteger;
begin
  V := TInteger.Create;
end.
```

Make sure the identifier really refers to an object or class type - maybe it is misspelled or it is hidden by an identifier from another unit.

3.1.2.1.337 E2254: Overloaded procedure '%s' must be marked with the 'overload' directive

The compiler has encountered a procedure, which is not marked overload, with the same name as a procedure already marked overload. All overloaded procedures must be marked as such.

```
program Produce;

procedure f0(a : integer); overload;
begin
end;

procedure f0(a : integer; ch : char);
begin
end;

begin
end.
```

The procedure f0(a : integer; ch : char) causes the error since it is not marked with the overload keyword.

```
program solve;

procedure f0(a : integer); overload;
begin
end;

procedure f0(a : integer; ch : char); overload;
begin
end;

begin
end.
```

If the procedure is intended to be an overloaded version, then mark it as overload. If it is not intended to be an overloaded version, then change its name.

3.1.2.1.338 E2017: Pointer type required

This error message is given when you apply the dereferencing operator '^' to an operand that is not a pointer, and, as a very special case, when the second operand in a 'Raise <exception> at <address>' statement is not a pointer.

```
program Produce;
var
  C: TObject;
begin
  C^.Destroy;
end.
```

Even though class types are implemented internally as pointers to the actual information, it is illegal to apply the dereferencing operator to class types at the source level. It is also not necessary - the compiler will dereference automatically whenever it is appropriate.

```
program Solve;
var
  C: TObject;
begin
  C.Destroy;
end.
```

Simply leave off the dereferencing operator—the compiler will do the right thing automatically.

3.1.2.1.339 E2267: Previous declaration of '%s' was not marked with the 'overload' directive

There are two solutions to this problem. You can either remove the attempt at overloading or you can mark the original declaration with the overload directive. The example shown here marks the original declaration.

```
program Produce;
type
  Base = class
    procedure func(a : integer);
    procedure func(a : char); overload;
  end;

  procedure Base.func(a : integer);
  begin
  end;

  procedure Base.func(a : char);
  begin
  end;

end.
```

This example attempts to overload the char version of func without marking the first version of func as overloadable.

You must mark all functions to be overloaded with the overload directive. If overload were not required on all versions it would be possible to introduce a new method which overloads an existing method and then a simple recompilation of the source could produce different behavior.

```
program Solve;
type
  Base = class
    procedure func(a : integer); overload;
    procedure func(a : char); overload;
  end;

  procedure Base.func(a : integer);
  begin
  end;

  procedure Base.func(a : char);
  begin
  end;

end.
```

3.1.2.1.340 E2121: Procedure or function name expected

You have specified an identifier which does not represent a procedure or function in an EXPORTS clause.

```
library Produce;

  var
   y : procedure;

exports y;
begin
end.
```

It is not possible to export variables from a built-in library, even though the variable is of 'procedure' type.

```
program Solve;
```

```
  procedure ExportMe;
  begin
  end;

exports ExportMe;
begin
end.
```

Always be sure that all the identifiers listed in an EXPORTS clause truly represent procedures.

3.1.2.1.341 E2299: Property required

You need to add a property to your program.

The declaration of a property specifies a name and a type, and includes at least one access specifier. The syntax of a property declaration is:

```
property propertyName[indexes]: type index integerConstant specifiers;
```

where:

— propertyName is any valid identifier

— [indexes] is optional and is a sequence of parameter declarations separated by semicolons

— Each parameter declaration has the form identifier1, ..., identifiern: type

— type must be a predefined or previously declared type identifier. That is, property declarations like property Num: 0..9 ... are invalid.

— the index integerConstant clause is optional.

— specifiers is a sequence of read, write, stored, default (or nodefault), and implements specifiers.

Every property declaration must have at least one read or write specifier.

For more information, see the 'About Properties' topic in the on-line Help.

3.1.2.1.342 E2018: Record, object or class type required

The compiler was expecting to find the type name which specified a record, object or class but did not find one.

```
  program Produce;

    type
      RecordDesc = class
        ch : Char;
      end;

    var
      pCh : PChar;
      r : RecordDesc;

    procedure A;
    begin
      pCh.ch := 'A';    (* case 1 *)

      with pCh do begin (* case 2 *)
      end;
    end;
  end.
```

There are two causes for the same error in this program. The first is the application of '.' to a object that is not a record. The second case is the use of a variable which is of the wrong type in a WITH statement.

```
  program Solve;

    type
      RecordDesc = class
        ch : Char;
      end;

    var
      r : RecordDesc;

    procedure A;
    begin
      r.ch := 'A';    (* case 1 *)

      with r do begin (* case 2 *)
      end;
    end;
  end.
```

The easy solution to this error is to always make sure that the '.' and WITH are both applied only to records, objects or class variables.

3.1.2.1.343 E2023: Function needs result type

You have declared a function, but have not specified a return type.

```
program Produce;

function Sum(A: array of Integer);
var I: Integer;
begin
  Result := 0;
  for I := 0 to High(A) do
    Result := Result + A[I];
end;

begin
end.
```

Here Sum is meant to be function, we have not told the compiler about it.

```
program Solve;

function Sum(A: array of Integer): Integer;
var I: Integer;
begin
  Result := 0;
  for I := 0 to High(A) do
    Result := Result + A[I];
end;

begin
end.
```

Just make sure you specify the result type.

3.1.2.1.344 E2366: Global procedure or class static method expected

No further information is available for this error or warning.

3.1.2.1.345 E2036: Variable required

This error message occurs when you try to take the address of an expression or a constant.

```
program Produce;
var
  I: Integer;
  PI: ^Integer;
begin
  PI := Addr(1);
end.
```

A constant like 1 does not have a memory address, so you cannot apply the operator or the Addr standard function to it.

```
program Solve;
var
  I: Integer;
  PI: ^Integer;
begin
  PI := Addr(I);
end.
```

You need to make sure you take the address of variable.

3.1.2.1.346 E2082: TYPEOF can only be applied to object types with a VMT

This error message is issued if you try to apply the standard function TypeOf to an object type that does not have a virtual method table.

A simple workaround is to declare a dummy virtual procedure to force the compiler to generate a VMT.

```
program Produce;

type
  TMyObject = object
    procedure MyProc;
  end;

procedure TMyObject.MyProc;
begin
  (*...*)
end;

var
  P: Pointer;
begin
  P := TypeOf(TMyObject);    (*<-- Error message here*)
end.
```

The example tries to apply the TypeOf standard function to type TMyObject which does not have virtual functions, and therefore no virtual function table (VMT).

```
program Solve;

type
  TMyObject = object
    procedure MyProc;
    procedure Dummy; virtual;
  end;

procedure TMyObject.MyProc;
begin
  (*...*)
end;
```

```
procedure TMyObject.Dummy;
begin
end;

var
  P: Pointer;
begin
  P := TypeOf(TMyObject);
end.
```

The solution is to introduce a dummy virtual function, or to eliminate the call to TypeOf.

3.1.2.1.347 E2014: Statement expected, but expression of type '%s' found

The compiler was expecting to find a statement, but instead it found an expression of the specified type.

```
  program Produce;
    var
      a : Integer;
  begin
     (3 + 4);
  end.
```

In this example, the compiler is expecting to find a statement, such as an IF, WHILE, REPEAT, but instead it found the expression (3+4).

```
  program Produce;
    var
      a : Integer;
  begin
     a := (3 + 4);
  end.
```

The solution here was to assign the result of the expression (3+4) to the variable 'a'. Another solution would have been to remove the offending expression from the source code - the choice depends on the situation.

3.1.2.1.348 E2279: Too many nested conditional directives

Conditional-directive constructions can be nested up to 32 levels deep.

3.1.2.1.349 E2409: Fully qualified nested type name %s exceeds 1024 byte limit

No further information is available for this error or warning.

3.1.2.1.350 E2079: Procedure NEW needs constructor

This error message is issued when an identifier given in the parameter list to New is not a constructor.

```
program Produce;

type
  PMyObject = ^TMyObject;
  TMyObject = object
  F: Integer;
  constructor Init;
  destructor Done;
  end;

constructor TMyObject.Init;
```

```
begin
  F := 42;
end;

destructor TMyObject.Done;
begin
end;

var
  P: PMyObject;

begin
  New(P, Done);              (*<-- Error message here*)
end.
```

By mistake, we called New with the destructor, not the constructor.

```
program Solve;

type
  PMyObject = ^TMyObject;
  TMyObject = object
  F: Integer;
  constructor Init;
  destructor Done;
  end;

constructor TMyObject.Init;
begin
  F := 42;
end;

destructor TMyObject.Done;
begin
end;

var
  P: PMyObject;

begin
  New(P, Init);
end.
```

Make sure you give the New standard function a constructor, or no additional argument at all.

3.1.2.1.351 W1039: No configuration files found

The compiler could not locate the configuration files referred to in the source code.

3.1.2.1.352 E2256: Dispose not supported (nor necessary) for dynamic arrays

The compiler has encountered a use of the standard procedure DISPOSE on a dynamic array. Dynamic arrays are reference counted and will automatically free themselves when there are no longer any references to them.

```
program Produce;
  var
    arr : array of integer;

begin
  SetLength(arr, 10);
  Dispose(arr);
end.
    The use of DISPOSE on the dynamic array arr causes the error in this example.
```

```
program Produce;
  var
    arr : array of integer;

begin
  SetLength(arr, 10);
end.
```

The only solution here is to remove the offending use of DISPOSE

3.1.2.1.353 E2250: There is no overloaded version of '%s' that can be called with these arguments

An attempt has been made to call an overloaded function that cannot be resolved with the current set of overloads.

```
program Produce;

procedure f0(a : integer); overload;
begin
end;

procedure f0(a : char); overload;
begin
end;

begin
  f0(1.2);
end.
```

The overloaded procedure f0 has two versions: one which takes a char and one which takes an integer. However, the call to f0 uses a floating point type, which the compiler cannot resolve into neither a char nor an integer.

```
program Solve;

procedure f0(a : integer); overload;
begin
end;

procedure f0(a : char); overload;
begin
end;

begin
  f0(1);
end.
```

You can solve this problem in two ways: either supply a parameter type which can be resolved into a match of an overloaded procedure, or create a new version of the overloaded procedure which matches the parameter type.

In the example above, the parameter type has been modified to match one of the existing overloaded versions of f0.

3.1.2.1.354 E2450: There is no overloaded version of array property '%s' that can be used with these arguments

To correct this error, either change the arguments so that their types match a version of the array property, or add a new overload of the array property with types that match the arguments.

3.1.2.1.355 E2273: No overloaded version of '%s' with this parameter list exists

An attempt has been made to call an overloaded procedure but no suitable match could be found.

```
program overload;
  procedure f(x : Char); overload;
  begin
  end;

  procedure f(x : Integer); overload;
  begin
  end;

begin
  f(1.0);

end.
```

In the use of f presented here, the compiler is unable to find a suitable match (using the type compatibility & overloading rules) given the actual parameter 1.0.

```
program overload;
  procedure f(x : char); overload;
  begin
  end;

  procedure f(x : integer); overload;
  begin
  end;

begin
  f(1);
end.
```

Here, the call to f has been changed to pass an integer as the actual parameter which will allow the compiler to find a suitable match. Another approach to solving this problem would be to introduce a new procedure which takes a floating point parameter.

3.1.2.1.356 E2025: Procedure cannot have a result type

You have declared a procedure, but given it a result type. Either you really meant to declare a function, or you should delete the result type.

```
program Produce;

procedure DotProduct(const A,B: array of Double): Double;
var
  I: Integer;
begin
  Result := 0.0;
  for I := 0 to High(A) do
    Result := Result + A[I]*B[I];
end;

const
  C: array [1..3] of Double = (1,2,3);

begin
  Writeln( DotProduct(C,C) );
end.
```

Here DotProduct was really meant to be a function, we just happened to use the wrong keyword...

```
program Solve;

function DotProduct(const A,B: array of Double): Double;
var
  I: Integer;
begin
  Result := 0.0;
  for I := 0 to High(A) do
    Result := Result + A[I]*B[I];
end;

const
  C: array [1..3] of Double = (1,2,3);

begin
  Writeln( DotProduct(C,C) );
end.
```

Just make sure you specify a result type when you declare a function, and no result type when you declare a procedure.

3.1.2.1.357 W1035: Return value of function '%s' might be undefined

This warning is displayed if the return value of a function has not been assigned a value on every code path.

To put it another way, the function could execute so that it never assigns anything to the return value.

```
program Produce;
(*$WARNINGS ON*)
var
  B: Boolean;
  C: (Red,Green,Blue);

function Simple: Integer;
begin
end;                         (*<-- Warning here*)

function IfStatement: Integer;
begin
  if B then
    Result := 42;
end;                           (*<-- Warning here*)

function CaseStatement: Integer;
begin
  case C of
  Red..Blue: Result := 42;
  end;
end;                           (*<-- Warning here*)

function TryStatement: Integer;
begin
  try
    Result := 42;
  except
    Writeln('Should not get here!');
  end;
end;                           (*<-- Warning here*)

begin
  B := False;
end.
```

The problem with procedure IfStatement and CaseStatement is that the result is not assigned in every code path. In TryStatement, the compiler assumes that an exception could happen before Result is assigned.

```
program Solve;
(*$WARNINGS ON*)
var
  B: Boolean;
  C: (Red,Green,Blue);

function Simple: Integer;
begin
  Result := 42;
end;

function IfStatement: Integer;
begin
  if B then
    Result := 42
  else
    Result := 0;
end;

function CaseStatement: Integer;
begin
  case C of
  Red..Blue: Result := 42;
  else       Result := 0;
  end;
end;

function TryStatement: Integer;
begin
  Result := 0;
  try
    Result := 42;
  except
    Writeln('Should not get here!');
  end;
end;

begin
  B := False;
end.
```

The solution is to make sure there is an assignment to the result variable in every possible code path.

3.1.2.1.358 E2134: Type '%s' has no type info

You have applied the TypeInfo standard procedure to a type identifier which does not have any run-time type information associated with it.

```
program Produce;

  type
    Data = record
    end;

  var
    v : Pointer;

begin
  v := TypeInfo(Data);
end.
```

Record types do not generate type information, so this use of TypeInfo is illegal.

```
program Solve;
```

```
    type
      Base = class
      end;

    var
      v : Pointer;

  begin
    v := TypeInfo(Base);
  end.
```

A class does generate RTTI, so the use of TypeInfo here is perfectly legal.

3.1.2.1.359 E2220: Never-build package '%s' requires always-build package '%s'

You are attempting to create a no-build package which requires an always-build package. Since the interface of an always-build package can change at anytime, and since giving the no-build flag instructs the compiler to assume that a package is up-to-date, each no-build package can only require other packages that are also marked no-build.

```
package Base;
end.

(*$IMPLICITBUILD OFF*)
package NoBuild;
  requires Base;
end.
```

In this example, the NoBuild package requires a package which was compiled in the always-build compiler state.

```
(*$IMPLICITBUILD OFF*)
package Base;
end.

(*$IMPLICITBUILD OFF*)
package NoBuild;
  requires Base;
end.
```

The solution used in this example was to turn Base into a never-build package. Another viable option would have been to remove the (*$IMPLICITBUILD OFF*) from the NoBuild package, thereby turning it into an always-build package.

3.1.2.1.360 E2093: Label '%s' is not declared in current procedure

In contrast to standard Pascal, Borland's Delphi language does not allow a goto to jump out of the current procedure.

However, his construct is mainly useful for error handling, and the Delphi language provides a more general and structured mechanism to deal with errors: exception handling.

```
program Produce;

label 99;

procedure MyProc;
begin
  (*Something goes very wrong...*)
  goto 99;
end;

begin
  MyProc;
  99:
    Writeln('Fatal error');
end.
```

The example above tries to halt computation by doing a non-local goto.

```
program Solve;

uses SysUtils;

procedure MyProc;
begin
  (*Something goes very wrong...*)
  raise Exception.Create('Fatal error');
end;

begin
  try
    MyProc;
  except
    on E: Exception do Writeln(E.Message);
  end;
end.
```

In our solution, we used exception handling to stop the program. This has the advantage that we can also pass an error message. Another solution would be to use the standard procedures Halt or RunError.

3.1.2.1.361 x2269: Overriding virtual method '%s.%s' has lower visibility (%s) than base class '%s' (%s)

The method named in the error message has been declared as an override of a virtual method in a base class, but the visibility in the current class is lower than that used in the base class for the same method.

While the visibility rules of Delphil would seem to indicate that the function cannot be seen, the rules of invoking virtual functions will cause the function to be properly invoked through a virtual call.

Generally, this means that the method of the derived class was declared in a private or protected section while the method of the base class was declared in a protected or pubic (including published) section, respectively.

```
unit Produce;
interface

  type
    Base = class(TObject)
    public
      procedure VirtualProcedure(X: Integer); virtual;
    end;

    Extended = class(Base)
    protected
      procedure VirtualProcedure(X: Integer); override;
    end;

implementation

  procedure Base.VirtualProcedure(X: Integer);
  begin
  end;

  procedure Extended.VirtualProcedure(X: Integer);
  begin
  end;
end.
```

The example above aptly shows how this error is produced by putting Extended.VirtualProcedure into the protected section.

In practice this is never harmful, but it can be confusing to someone reading documentation and observing the visibility attributes of the document.

This hint will only be produced for classes appearing in the interface section of a unit.

```
unit Solve;
interface

  type
    Base = class(TObject)
    public
      procedure VirtualProcedure(X: Integer); virtual;
    end;

    Extended = class(Base)
    public
      procedure VirtualProcedure(X: Integer); override;
    end;

implementation

  procedure Base.VirtualProcedure(X: Integer);
  begin
  end;

  procedure Extended.VirtualProcedure(X: Integer);
  begin
  end;
end.
```

There are three basic solutions to this problem.

1. Ignore the hint
2. Change the visibility to match the base class
3. Move class definition to the implementation section.

The example program above has taken the approach of changing the visibility to match the base class.

3.1.2.1.362 E2411: Unit %s in package %s refers to unit %s which is not found in any package. Packaged units must refer only to packaged units

No further information is available for this error or warning.

3.1.2.1.363 E2236: Constructors and destructors must have %s calling convention

An attempt has been made to change the calling convention of a constructor or destructor from the default calling convention.

```
program Produce;

  type
    TBase = class
      constructor Create; pascal;
    end;

  constructor TBase.Create;
  begin
  end;

begin
end.
```

```
    program Solve;

  type
    TBase = class
      constructor Create;
    end;

  constructor TBase.Create;
  begin
  end;

begin
end.
```

The only viable approach when this error has been issued by the compiler is to remove the offending calling convention directive from the constructor or destructor definition, as has been done in this example.

3.1.2.1.364 E2179: Only register calling convention allowed in OLE automation section

You have specified an illegal calling convention on a method appearing in an 'automated' section of a class declaration.

```
program Produce;

  type
    Base = class
    automated
      procedure Method; cdecl;
    end;

  procedure Base.Method; cdecl;
  begin
  end;

begin
end.
```

The language specification disallows all calling conventions except 'register' in an OLE automation section. The offending statement is 'cdecl' in the above code.

```
program Solve;

  type
    Base = class
    automated
      procedure Method; register;
      procedure Method2;
    end;

  procedure Base.Method; register;
  begin
  end;

  procedure Base.Method2;
  begin
  end;

begin
end.
```

There are three solutions to this error. The first is to specify no calling convention on methods declared in an auto section. The second is to specify only the register calling convention. The third is to move the offending declaration out of the automation section.

3.1.2.1.365 E2270: Published property getters and setters must have %s calling convention

A property appearing in a published section has a getter or setter procedure that does not have the correct calling convention.

```
unit Produce;
interface
  type
    Base = class
    public
      function getter : Integer; cdecl;
    published
      property Value : Integer read getter;
    end;

implementation
function Base.getter : Integer;
begin getter := 0;
end;

end.
```

This example declares the getter function getter for the published property Value to be of cdecl calling convention, which produces the error.

```
unit Solve;
interface
  type
    Base = class
    public
      function getter : Integer;
    published
      property Value : Integer read getter;
    end;

implementation
function Base.getter : Integer;
begin getter := 0;
end;

end.
```

The only solution to this problem is to declare the getter function to match the correct calling convention, which is the default. As can be seen in this example, no calling convention is specified.

3.1.2.1.366 E2391: Potentially polymorphic constructor calls must be virtual

No further information is available for this error or warning.

3.1.2.1.367 E2242: '%s' is not the name of a unit

The $NOINCLUDE directive must be given a known unit name.

3.1.2.1.368 E2064: Left side cannot be assigned to

This error message is given when you try to modify a read-only object like a constant, a constant parameter, or the return value of function.

```
program Produce;
```

```
const
  c = 1;

procedure p(const s: string);
begin
  s := 'changed';            (*<-- Error message here*)
end;

function f: PChar;
begin
  f := 'Hello';              (*This is fine - we are setting the return value*)
end;

begin
  c := 2;                    (*<-- Error message here*)
  f := 'h';                  (*<-- Error message here*)
end.
```

The example assigns to constant parameter, to a constant, and to the result of a function call. All of these are illegal.

```
program Solve;

var
  c : Integer = 1;           (*Use an initialized variable*)

procedure p(var s: string);
begin
  s := 'changed';            (*Use variable parameter*)
end;

function f: PChar;
begin
  f := 'Hello';              (*This is fine - we are setting the return value*)
end;

begin
  c := 2;
  f^ := 'h';                 (*This compiles, but will crash at runtime*)
end.
```

There two ways you can solve this kind of problem: either you change the definition of whatever you are assigning to, so the assignment becomes legal, or you eliminate the assignment.

3.1.2.1.369 E2430: for-in statement cannot operate on collection type '%s'

A for-in statement can only operate on the following collection types:

- Primitive types that the compiler recognizes, such as arrays, sets or strings
- Types that implement IEnumerable
- Types that implement the GetEnumerator pattern as documented in the Delphi Language Guide

Ensure that the specified type meets these requirements.

See Also

Declarations and Statements (see page 705)

3.1.2.1.370 H2135: FOR or WHILE loop executes zero times - deleted

The compiler has determined that the specified looping structure will not ever execute, so as an optimization it will remove it. Example:

```
program Produce;
(*$HINTS ON*)

  var
    i : Integer;

begin
  i := 0;
  WHILE FALSE AND (i < 100) DO
    INC(i);
end.
```

The compiler determines that 'FALSE AND (i < 100)' always evaluates to FALSE, and then easily determines that the loop will not be executed.

```
program Solve;
(*$HINTS ON*)

  var
    i : Integer;

begin
  i := 0;
  WHILE i < 100 DO
    INC(i);
end.
```

The solution to this hint is to check the boolean expression used to control while statements is not always FALSE. In the for loops you should make sure that (upper bound - lower bound) >= 1.

You may see this warning if a FOR loop increments its control variable from a value within the range of Longint to a value outside the range of Longint. For example:

```
var I: Cardinal;
begin
  For I := 0 to $FFFFFFFF do
...
```

This results from a limitation in the compiler which you can work around by replacing the FOR loop with a WHILE loop.

3.1.2.1.371 E2248: Cannot use old style object types when compiling to byte code

Old-style Object types are illegal when compiling to byte code.

3.1.2.1.372 E2058: Class, interface and object types only allowed in type section

Class or object types must always be declared with an explicit type declaration in a type section - unlike record types, they cannot be anonymous.

The main reason for this is that there would be no way you could declare the methods of that type - after all, there is no type name.

```
program Produce;

var
  MyClass : class
    Field: Integer;
  end;
```

```
begin
end.
```

The example tries to declare a class type within a variable declaration - that is not legal.

```
program Solve;

type
  TMyClass = class
    Field: Integer;
  end;

var
  MyClass : TMyClass;

begin
end.
```

The solution is to introduce a type declaration for the class type. Alternatively, you could have changed the class type to a record type.

3.1.2.1.373 E2059: Local class, interface or object types not allowed

Class and object cannot be declared local to a procedure.

```
program Produce;

  procedure MyProc;
  type
    TMyClass = class
      Field: Integer;
    end;
  begin
  (*...*)
  end;

begin
end.
```

So MyProc tries to declare a class type locally, which is illegal.

```
program Solve;

  type
    TMyClass = class
      Field: Integer;
    end;

  procedure MyProc;
  begin
  (*...*)
  end;

begin
end.
```

The solution is to move out the declaration of the class or object type to the global scope.

3.1.2.1.374 E2062: Virtual constructors are not allowed

Unlike class types, object types can only have static constructors.

```
program Produce;
```

```
type
  TMyObject = object
    constructor Init; virtual;
  end;

constructor TMyObject.Init;
begin
end;

begin
end.
```

The example tries to declare a virtual constructor, which does not really make sense for object types and is therefore illegal.

```
program Solve;

type
  TMyObject = object
    constructor Init;
  end;

constructor TMyObject.Init;
begin
end;

begin
end.
```

The solution is to either make the constructor static, or to use a new-style class type which can have a virtual constructor.

3.1.2.1.375 E2439: Inline function must not have open array argument

To avoid this error, remove the **inline** directive or use an explicitly-declared dynamic array type instead of an open array argument.

See Also

Parameters (↗ see page 672)

Structured Types (↗ see page 566)

3.1.2.1.376 W1049: value '%s' for option %s was truncated

String based compiler options such as unit search paths have finite buffer limits.

This message indicates you have exceeded the buffer limit.

3.1.2.1.377 E2001: Ordinal type required

The compiler required an ordinal type at this point. Ordinal types are the predefined types Integer, Char, WideChar, Boolean, and declared enumerated types.

Ordinal types are required in several different situations:

- The index type of an array must be ordinal.
- The low and high bounds of a subrange type must be constant expressions of ordinal type.
- The element type of a set must be an ordinal type.
- The selection expression of a case statement must be of ordinal type.
- The first argument to the standard procedures Inc and Dec must be a variable of either ordinal or pointer type.

```
program Produce;
type
TByteSet = set of 0..7;
var
BitCount: array [TByteSet] of Integer;
begin
end.
```

The index type of an array must be an ordinal type - type TByteSet is a set, not an ordinal.

```
program Solve;
type
  TByteSet = set of 0..7;
var
  BitCount: array [Byte] of Integer;
begin
end.
```

Supply an ordinal type as the array index type.

3.1.2.1.378 E2271: Property getters and setters cannot be overloaded

A property has specified an overloaded procedure as either its getter or setter.

```
unit Produce;
interface
  type
    Base = class
    public
      function getter : Integer; overload;
      function getter(a : char) : Integer; overload;
      property Value : Integer read getter;
    end;

implementation
function Base.getter : Integer;
begin getter := 0;
end;

function Base.getter(a : char) : Integer;
begin
end;

end.
```

The overloaded function getter in the above example will cause this error.

```
unit Solve;
interface
  type
    Base = class
    public
      function getter : Integer;
      property Value : Integer read getter;
    end;

implementation
function Base.getter : Integer;
begin getter := 0;
end;

end.
```

The only solution when this problem occurs is to remove the offending overload specifications, as is shown in the above example.

3.1.2.1.379 H2365: Override method %s.%s should match case of ancestor %s.%s

No further information is available for this error or warning.

3.1.2.1.380 E2137: Method '%s' not found in base class

You have applied the 'override' directive to a method, but the compiler is unable to find a procedure of the same name in the base class.

```
program Produce;

  type
    Base = class
      procedure Title; virtual;
    end;

    Derived = class (Base)
      procedure Titl; override;
    end;

    procedure Base.Title;
    begin
    end;

    procedure Derived.Titl;
    begin
    end;

begin
end.
```

A common cause of this error is a simple typographical error in your source code. Make sure that the name used as the 'override' procedure is spelled the same as it is in the base class. In other situations, the base class will not provide the desired procedure: it is those situations which will require much deeper analysis to determine how to solve the problem.

```
program Solve;

  type
    Base = class
      procedure Title; virtual;
    end;

    Derived = class (Base)
      procedure Title; override;
    end;

    procedure Base.Title;
    begin
    end;

    procedure Derived.Title;
    begin
    end;

begin
end.
```

The solution (in this example) was to correct the spelling of the procedure name in Derived.

3.1.2.1.381 E2352: Cannot override a final method

No further information is available for this error or warning.

3.1.2.1.382 E2170: Cannot override a non-virtual method

You have tried, in a derived class, to override a base method which was not declared as one of the virtual types.

```
program Produce;

  type
    Base = class
      procedure StaticMethod;
    end;

    Derived = class (Base)
      procedure StaticMethod; override;
    end;

    procedure Base.StaticMethod;
    begin
    end;

    procedure Derived.StaticMethod;
    begin
    end;

begin
end.
```

The example above elicits an error because Base.StaticMethod is not declared to be a virtual method, and as such it is not possible to override its declaration.

```
program Solve;

  type
    Base = class
      procedure StaticMethod;
    end;

    Derived = class (Base)
      procedure StaticMethod;
    end;

    procedure Base.StaticMethod;
    begin
    end;

    procedure Derived.StaticMethod;
    begin
    end;

begin
end.
```

The only way to remove this error from your program, when you don't have the source for the base classes, is to remove the 'override' specification from the declaration of the derived method. If you have source to the base classes, you could, with careful consideration, change the base's method to be declared as one of the virtual types. Be aware, however, that this change can have a drastic affect on your programs.

3.1.2.1.383 F2220: Could not compile package '%s'

An error occurred while trying to compile the package named in the message. The only solution to the problem is to correct the error and recompile the package.

3.1.2.1.384 E2199: Packages '%s' and '%s' both contain unit '%s'

The project you are trying to compile is using two packages which both contain the same unit. It is illegal to have two packages which are used in the same project containing the same unit since this would cause an ambiguity for the compiler.

A main cause of this problem is a poorly defined package set.

The only solution to this problem is to redesign your package hierarchy to remove the ambiguity.

3.1.2.1.385 E2200: Package '%s' already contains unit '%s'

The package you are compiling requires (either through the requires clause or the package list) another package which already contains the unit specified in the message.

It is an error to have to related packages contain the same unit. The solution to this problem is to remove the unit from one of the packages or to remove the relation between the two packages.

3.1.2.1.386 W1031: Package '%s' will not be written to disk because -J option is enabled

The compiler can't write the package to disk because the -J option is attempting to create an object file.

3.1.2.1.387 E2225: Never-build package '%s' must be recompiled

The package referenced in the message was compiled as a never-build package, but it requires another package to which interface changes have been made. The named package cannot be used without recompiling because it was linked with a different interface of the required package.

The only solution to this error is to manually recompile the offending package. Be sure to specify the never-build switch, if it is still desired.

3.1.2.1.388 H2235: Package '%s' does not use or export '%s.%s'

You have compiled a unit into a package which contains a symbol which does not appear in the interface section of the unit, nor is it referenced by any code in the unit. In effect, this code is dead code and could be removed from the unit without changing the semantics of your program.

3.1.2.1.389 E2201: Need imported data reference ($G) to access '%s' from unit '%s'

The unit named in the message was not compiled with the $G switch turned on.

```
(*$IMPORTEDDATA OFF*)
unit u0;
interface
```

```
implementation
begin
  WriteLn(System.RandSeed);
end.

program u1;
  uses u0;
end.
```

In the above example, u0 should be compiled alone. Then, u1 should be compiled with CLXxx (where xx represents the version). The problem occurs because u0 is compiled under the premise that it will never use data which resides in a package.

```
(*$IMPORTEDDATA ON*)
unit u0;
interface
implementation
begin
  WriteLn(System.RandSeed);
end.

program u1;
  uses u0;
end.
```

To alleviate the problem, it is generally easiest to turn on the $IMPORTEDDATA switch and recompile the unit that produces the error.

3.1.2.1.390 W1032: Exported package threadvar '%s.%s' cannot be used outside of this package

Windows does not support the exporting of threadvar variables from a DLL, but since using packages is meant to be semantically equivalent to compiling a project without them, the Delphi compiler must somehow attempt to support this construct.

This warning is to notify you that you have included a unit which contains a threadvar in an interface into a package. While this is not illegal, you will not be able to access the variable from a unit outside the package.

Attempting to access this variable may appear to succeed, but it actually did not.

A solution to this warning is to move the threadvar to the implementation section and provide function which will retrieve the variables value.

3.1.2.1.391 E2213: Bad packaged unit format: %s.%s

When the compiler attempted to load the specified unit from the package, it was found to be corrupt. This problem could be caused by an abnormal termination of the compiler when writing the package file (for example, a power loss). The first recommended action is to delete the offending DCP file and recompile the package.

3.1.2.1.392 E2006: PACKED not allowed here

The packed keyword is only legal for set, array, record, object, class and file types. In contrast to the 16-bit version of Delphi, packed will affect the layout of record, object and class types.

```
program Produce;
type
  SmallReal = packed Real;
begin
end.
```

Packed can not be applied to a real type - if you want to conserve storage, you need to use the smallest real type, type Single.

```
program Solve;
type
  SmallReal = Single;
begin
end.
```

3.1.2.1.393 E2394: Parameterless constructors not allowed on record types

No further information is available for this error or warning.

3.1.2.1.394 E2363: Only methods of descendent types may access protected symbol [%s]%s.%s across assembly boundaries

No further information is available for this error or warning.

3.1.2.1.395 E2375: PRIVATE or PROTECTED expected

No further information is available for this error or warning.

3.1.2.1.396 W1045: Property declaration references ancestor private '%s.%s'

This warning indicates that your code is not portable to C++. This is important for component writers who plan to distribute custom components.

In the Delphi language, you can declare a base class with a private member, and a child class in the same unit can refer to the private member. In C++, this construction is not permitted. To fix it, change the child to refer to either a protected member of the base class or a protected member of the child class.

Following is an example of code that would cause this error:

```
type
      TBase = class(…)
            private
        FFoo:Integer
            end;
      TChild=class(TBase)
            published
            property foo:Integer read FFoo write FFoo;
            end;
```

3.1.2.1.397 H2219: Private symbol '%s' declared but never used

The symbol referenced appears in a private section of a class, but is never used by the class. It would be more memory efficient if you removed the unused private field from your class definition.

```
program Produce;
  type
    Base = class
    private
      FVar : Integer;
      procedure Init;
    end;

procedure Base.Init;
begin
end;

begin
```

```
end.
```

Here we have declared a private variable which is never used. The message will be emitted for this case.

```
program Solve;
program Produce;
  type
    Base = class
    private
      FVar : Integer;
      procedure Init;
    end;

procedure Base.Init;
begin
  FVar := 0;
end;

begin
end.
```

There are various solutions to this problem, and since this message is not an error message, all are correct. If you have included the private field for some future use, it would be valid to ignore the message. Or, if the variable is truly superfluous, it can be safely removed. Finally, it might have been a programming oversight not to use the variable at all; in this case, simply add the code you forgot to implement.

3.1.2.1.398 E2357: PROCEDURE, FUNCTION, or CONSTRUCTOR expected

No further information is available for this error or warning.

3.1.2.1.399 E2122: PROCEDURE or FUNCTION expected

This error message is produced by two different constructs, but in both cases the compiler is expecting to find the keyword 'procedure' or the keyword 'function'.

```
program Produce;

  type
    Base = class
      class AProcedure; (*case 1*)
    end;

  class Base.AProcedure; (*case 2*)
  begin
  end;

begin
end.
```

In both cases above, the word 'procedure' should follow the keyword 'class'.

```
program Solve;

  type
    Base = class
      class procedure AProcedure;
    end;

  class procedure Base.AProcedure;
  begin
  end;

begin
```

```
end.
```

As can be seen, adding the keyword 'procedure' removes the error from this program.

3.1.2.1.400 x2367: Case of property accessor method %s.%s should be %s.%s

No further information is available for this error or warning.

3.1.2.1.401 E2300: Cannot generate property accessor '%s' because '%s' already exists

No further information is available for this error or warning.

3.1.2.1.402 E2370: Cannot use inherited methods for interface property accessors

No further information is available for this error or warning.

3.1.2.1.403 H2369: Property accessor %s should be %s

No further information is available for this error or warning.

3.1.2.1.404 H2368: Visibility of property accessor method %s should match property %s.%s

No further information is available for this error or warning.

3.1.2.1.405 E2181: Redeclaration of property not allowed in OLE automation section

It is not allowed to move the visibility of a property into an automated section.

```
program Produce;

  type
    Base = class
      v : Integer;
      s : String;
    protected
      property Name : String read s write s;
      property Value : Integer read v write v;
    end;

    Derived = class (Base)
    public
      property Name; (* Move Name to a public visibility by redeclaration *)
    automated
      property Value;
    end;

begin
end.
```

In the above example, Name is moved from a private visibility in Base to public visibility in Derived by redeclaration. The same idea is attempted on Value, but an error results.

```
program Solve;

  type
    Base = class
      v : Integer;
      s : String;
    protected
      property Name : String read s write s;
      property Value : Integer read v write v;
    end;

    Derived = class (Base)
    public
      property Name; (* Move Name to a public visibility by redeclaration *)
      property Value;
    automated
    end;

begin
end.
```

It is not possible to change the visibility of a property to an automated section, therefore the solution to this problem is to not redeclare properties of base classes in automated sections.

3.1.2.1.406 E2206: Property overrides not allowed in interface type

A property which was declared in a base interface has been overridden in an interface extension.

```
program Produce;
  type
    Base = interface
      function Reader : Integer;
      function Writer(a : Integer);
      property Value : Integer read Reader write Writer;
    end;

    Extension = interface (Base)
      function Reader2 : Integer;
      property Value Integer read Reader2;
    end;

begin
end.
```

The error in the example is that Extension attempts to override the Value property.

```
program Solve;
  type
    Base = interface
      function Reader : Integer;
      function Writer(a : Integer);
      property Value : Integer read Reader write Writer;
    end;

    Extension = interface (Base)
      function Reader2 : Integer;
      property Value2 Integer read Reader2;
    end;

begin
end.
```

A solution to this error is to rename the offending property. Another, more robust, approach is to determine the original intent and restructure the system design to solve the problem.

3.1.2.1.407 E2356: Property accessor must be an instance field or method

No further information is available for this error or warning.

3.1.2.1.408 E2434: Property declarations not allowed in anonymous record or local record type

Record types that are declared in local scopes or declared in-place in variable declarations can only contain field declarations. For advanced features in record types (such as methods, properties, and nested types), the record type must be an explicitly declared global type.

3.1.2.1.409 E2148: Dynamic method or message handler not allowed here

Dynamic and message methods cannot be used as accessor functions for properties.

```
program Produce;

  type
    Base = class
      v : Integer;
      procedure SetV(x : Integer); dynamic;
      function GetV : Integer; message;
      property Velocity : Integer read GetV write v;
      property Value : Integer read v write SetV;
    end;

  procedure Base.SetV(x : Integer);
  begin v := x;
  end;

  function Base.GetV : Integer;
  begin GetV := v;
  end;

begin
end.
```

Both 'Velocity' and 'Value' above are in error since they both have illegal accessor functions assigned to them.

```
program Solve;

  type
    Base = class
      v : Integer;
      procedure SetV(x : Integer);
      function GetV : Integer;
      property Velocity : Integer read GetV write v;
      property Value : Integer read v write SetV;
    end;

  procedure Base.SetV(x : Integer);
  begin v := x;
  end;

  function Base.GetV : Integer;
  begin GetV := v;
  end;
```

```
begin
end.
```

The solution taken in this is example was to remove the offending compiler directives from the procedure declarations; this may not be the right solution for you. You may have to closely examine the logic of your program to determine how best to provide accessor functions for your properties.

3.1.2.1.410 E2233: Property '%s' inaccessible here

An attempt has been made to access a property through a class reference type. It is not possible to access fields nor properties of a class through a class reference.

```
program Produce;

  type
    TBase = class
    public
      FX : Integer;
      property X : Integer read FX write FX;
    end;

    TBaseClass = class of TBase;

  var
    BaseRef : TBaseClass;
    x : Integer;

begin
  BaseRef := TBase;
  x := BaseRef.X;
end.
```

Attempting to access the property X in the example above causes the compiler to issue an error.

```
program Solve;

  type
    TBase = class
    public
      FX : Integer;
      property X : Integer read FX write FX;
    end;

    TBaseClass = class of TBase;

  var
    BaseRef : TBaseClass;
    x : Integer;

begin
  BaseRef := TBase;
end.
```

There is no other solution to this problem than to remove the offending property access from your source code. If you wish to access properties or fields of a class, then you need to create an instance variable of that class type and gain access through that variable.

3.1.2.1.411 E2275: property attribute 'label' cannot be an empty string

The error is output because the label attribute for g is an empty string.

```
unit Problem;
```

```
interface
  type
    T0 = class
      f : integer;
      property g : integer read f write f label '';
    end;

implementation
begin
end.
```

In this solution, the label attribute has been replaced by a non-zero length string.

```
unit Solve;
interface
  type
    T0 = class
      f : integer;
      property g : integer read f write f label 'LabelText';
    end;

implementation
begin
end.
```

3.1.2.1.412 E2292: '%s' must reference a property or field of class '%s'

In custom attribute declaration syntax, you can pass values to the constructor of the attribute class, followed by name=value pairs, where name is a property or field of the attribute class.

3.1.2.1.413 E2129: Cannot assign to a read-only property

The property to which you are attempting to assign a value did not specify a 'write' clause, thereby causing it to be a read-only property.

```
program Produce;

  type
    Base = class
      s : String;

      property Title : String read s;
    end;

  var
    c : Base;

  procedure DiddleTitle
  begin
    if c.Title = '' then
      c.Title := 'Super Galactic Invaders with Turbo Gungla Sticks';

      (*perform other work on the c.Title*)
  end;

begin
end.
```

If a property does not specify a 'write' clause, it effectively becomes a read-only property; it is not possible to assign a value to a property which is read-only, thus the compiler outputs an error on the assignment to 'c.Title'.

```
program Solve;

  type
```

```
    Base = class
      s : String;

      property Title : String read s;
    end;

  var
    c : Base;

  procedure DiddleTitle
    var title : String;
  begin
    title := c.Title;
    if Title = '' then
      Title := 'Super Galactic Invaders with Turbo Gungla Sticks';
      (*perform other work on title*)
  end;

begin
end.
```

One solution, if you have source code, is to provide a write clause for the read-only property - of course, this could dramatically alter the semantics of the base class and should not be taken lightly. Another alternative would be to introduce an intermediate variable which would contain the value of the read-only property - it is this second alternative which is shown in the code above.

3.1.2.1.414 E2130: Cannot read a write-only property

The property from which you are attempting to read a value did not specify a 'read' clause, thereby causing it to be a write-only property.

```
program Produce;

  type
    Base = class
      s : String;

      property Password : String write s;
    end;

  var
    c : Base;
    s : String;

begin
  s := c.Password;
end.
```

Since c.Password has not specified a read clause, it is not possible to read its value.

```
program Solve;

  type
    Base = class
      s : String;

      property Password : String read s write s;
    end;

  var
    c : Base;
    s : String;

begin
  s := c.Password;
end.
```

One easy solution to this problem, if you have source code, would be to add a read clause to the write-only property. But, adding a read clause is not always desirable and could lead to holes in a security system. Consider, for example, a write-only property called 'Password', as in this example: you certainly wouldn't want to casually allow programs using this class to read the stored password. If a property was created as write-only, there is probably a good reason for it and you should reexamine why you need to read this property.

3.1.2.1.415 E2362: Cannot access protected symbol %s.%s

No further information is available for this error or warning.

3.1.2.1.416 E2389: Protected member '%s' is inaccessible here

No further information is available for this error or warning.

3.1.2.1.417 H2244: Pointer expression needs no Initialize/Finalize - need ^ operator?

You have attempted to finalize a Pointer type.

```
program Produce;

  var
    str : String;
    pstr : PString;

begin
  str := 'Sharene';
  pstr := @str;
  Finalize(pstr);  (*note: do not attempt to use 'str' after this*)
end.
```

In this example the pointer, pstr, is passed to the Finalize procedure. This causes an hint since pointers do not require finalization.

```
program Solve;


  var
    str : String;
    pstr : PString;

begin
  str := 'Sharene';
  pstr := @str;
  Finalize(pstr^);  (*note: do not attempt to use 'str' after this*)
end.
```

The solution to this problem is to apply the ^ operator to the pointer which is passed to the Finalization procedure.

3.1.2.1.418 E2186: Published Real property '%s' must be Single, Real, Double or Extended

You have attempted to publish a property of type Real, which is not allowed. Published floating point properties must be Single, Double, or Extended.

```
program Produce;
  type
```

```
    Base = class
      R : Real48;
    published
      property RVal : Real read R write R;
    end;
end.
```

The published Real48 property in the program above must be either removed, moved to an unpublished section or changed into an acceptable type.

```
program Produce;
  type
    Base = class
      R : Single;
    published
      property RVal : Single read R write R;
    end;
end.
```

This solution changed the property into a real type that will actually produce run-time type information.

3.1.2.1.419 E2187: Size of published set '%s' is >4 bytes

The compiler does not allow sets greater than 32 bits to be contained in a published section. The size, in bytes, of a set can be calculated by High(setname) div 8 - Low(setname) div 8 + 1. -$M+

```
(*$TYPEINFO ON*)
program Produce;
  type
    CharSet = set of Char;
    NamePlate = class
      Characters : CharSet;
    published
      property TooBig : CharSet read Characters write Characters ;
    end;

begin
end.
    (*$TYPEINFO ON*)
program Solve;
  type
    CharSet = set of 'A'..'Z';
    NamePlate = class
      Characters : CharSet;
    published
      property TooBig : CharSet read Characters write Characters ;
    end;

begin
end.
```

3.1.2.1.420 E2361: Cannot access private symbol %s.%s

No further information is available for this error or warning.

3.1.2.1.421 E2390: Class must be sealed to call a private constructor without a type qualifier

No further information is available for this error or warning.

3.1.2.1.422 E2398: Class methods in record types must be static

No further information is available for this error or warning.

3.1.2.1.423 E2083: Order of fields in record constant differs from declaration

This error message occurs if record fields in a typed constant or initialized variable are not initialized in declaration order.

```
program Produce;

type
  TPoint = record
    X, Y: Integer;
  end;

var
  Point : TPoint = (Y: 123; X: 456);

begin
end.
```

The example tries to initialize first Y, then X, in the opposite order from the declaration.

```
program Solve;

type
  TPoint = record
    X, Y: Integer;
  end;

var
  Point : TPoint = (X: 456; Y: 123);

begin
end.
```

The solution is to adjust the order of initialization to correspond to the declaration order.

3.1.2.1.424 E2419: Record type too large: exceeds 1 MB

Records are limited to a size of 1MB according to the .NET SDK Documentation. Refer to Partition II Medatada 21.8 ClassLayout: 0x0F

3.1.2.1.425 E2245: Recursive include file %s

The $I directive has been used to recursively include another file. You must check to make sure that all include files terminate without having cycles in them.

3.1.2.1.426 F2092: Program or unit '%s' recursively uses itself

An attempt has been made for a unit to use itself.

```
unit Produce;
interface
  uses Produce;
implementation

begin
```

```
end.
```

In the above example, the uses clause specifies the same unit, which causes the compiler to emit an error message.

```
unit Solve;
interface
implementation

begin
end.
```

The only solution to this problem is to remove the offending uses clause.

3.1.2.1.427 E2214: Package '%s' is recursively required

When compiling a package, the compiler determined that the package requires itself.

```
package Produce;
  requires Produce;

end.
```

The error is caused because it is not legal for a package to require itself.

The only solution to this problem is to remove the recursive use of the package.

3.1.2.1.428 E2145: Re-raising an exception only allowed in exception handler

You have used the syntax of the raise statement which is used to reraise an exception, but the compiler has determined that this reraise has occurred outside of an exception handler block. A limitation of the current exception handling mechanism disallows reraising exceptions from nested exception handlers. for the exception.

```
program Produce;

  procedure RaiseException;
  begin
    raise;            (*case 1*)
    try
      raise;          (*case 2*)
    except
      try
        raise;        (*case 3*)
      except
      end;
      raise;
    end;
  end;


begin
end.
```

There are several reasons why this error might occur. First, you might have specified a raise with no exception constructor outside of an exception handler. Secondly, you might be attempting to reraise an exception in the try block of an exception handler. Thirdly, you might be attempting to reraise the exception in an exception handler nested in another exception handler.

```
program Solve;
  uses SysUtils;

  procedure RaiseException;
  begin
    raise Exception.Create('case 1');
    try
      raise Exception.Create('case 2');
    except
      try
        raise Exception.Create('case 3');
      except
      end;
      raise;
    end;
  end;

begin
end.
```

One solution to this error is to explicitly raise a new exception; this is probably the intention in situations like 'case 1' and 'case 2'. For the situation of 'case 3', you will have to examine your code to determine a suitable workaround which will provide the desired results.

3.1.2.1.429 E2377: Unable to locate Borland.Delphi.Compiler.ResCvt.dll

No further information is available for this error or warning.

3.1.2.1.430 E2381: Resource string length exceeds Windows limit of 4096 characters

No further information is available for this error or warning.

3.1.2.1.431 E2024: Invalid function result type

File types are not allowed as function result types.

```
program Produce;

function OpenFile(Name: string): File;
begin
end;

begin
end.
```

You cannot return a file from a function.

```
program Solve;

procedure OpenFile(Name: string; var F: File);
begin
end;

begin
end.
```

You can 'return' the file as a variable parameter. Alternatively, you can also allocate a file dynamically and return a pointer to it.

3.1.2.1.432 Linker error: %s

The resource linker (RLINK32) has encountered an error while processing a resource file. This error may be caused by any of the following reasons:

- You have used a duplicate resource name. Rename one of the resources.
- You have a corrupted resource file. You need to replace it with another version that is not corrupted or remove it.

- You are using an unsupported resource type, such as a 16-bit resource or form template.
- If converting resources such as 16-bit icons to 32-bit, the resource linker may have encountered problems.

3.1.2.1.433 Linker error: %s: %s

The resource linker (RLINK32) has encountered an error while processing a resource file. A resource linked into the project has the same type and name, or same type and resource ID, as another resource linked into the project. (In Delphi, duplicate resources are ignored with a warning. In Kylix, duplicates cause an error.)

3.1.2.1.434 E2215: 16-Bit segment encountered in object file '%s'

A 16-bit segment has been found in an object file that was loaded using the $L directive.

```
end.
```

The only solution to this error is to obtain an object file which does not have a 16-bit segment definition. You should consult the documentation for the product which produced the object file for instructions on turning 16-bit segment definitions into 32-bit segment definitions.

3.1.2.1.435 E2091: Segment/Offset pairs not supported in CodeGear 32-bit Pascal

32-bit code no longer uses the segment/offset addressing scheme that 16-bit code used.

In 16-bit versions of CodeGear Pascal, segment/offset pairs were used to declare absolute variables, and as arguments to the Ptr standard function.

Note that absolute addresses should not be used in 32-bit protected mode programs. Instead appropriate Win32 API functions should be called.

```
program Produce;

var
  VideoMode : Integer absolute $0040:$0049;

begin
  Writeln( Byte(Ptr($0040,$0049)^) );
end.
    program Solve;
(*This version will compile, but will not run; absolute addresses are to be carefully avoided*)
var
  VideoMode : Integer absolute $0040*16+$0049;

begin
  Writeln( Byte(Ptr($0040*16+$0049)^) );
end.
```

3.1.2.1.436 E2153: ';' not allowed before 'ELSE'

You have placed a ';' directly before an ELSE in an IF-ELSE statement. The reason for this is that the ';' is treated as a statement separator, not a statement terminator - IF-ELSE is one statement, a ';' cannot appear in the middle (unless you use compound statements).

```
program Produce;

  var
    b : Integer;
```

```
begin
  if b = 10 then
    b := 0;
  else
    b := 10;
end.
```

The Delphi language does not allow a ';' to be placed directly before an ELSE statement. In the code above, an error will be flagged because of this fact.

```
program Solve;

  var
    b : Integer;

begin
  if b = 10 then
    b := 0
  else
    b := 10;

  if b = 10 then begin
    b := 0;
  end
  else begin
    b := 10;
  end;

end.
```

There are two easy solutions to this problem. The first is to remove the offending ';'. The second is to create compound statements for each part of the IF-ELSE. If $HINTS are turned on, you will receive a hint about the value assigned to 'b' is never used.

3.1.2.1.437 E2028: Sets may have at most 256 elements

This error message appears when you try to declare a set type of more than 256 elements. More precisely, the ordinal values of the upper and lower bounds of the base type must be within the range 0..255.

```
program Produce;
type
  BigSet = set of 1..256;  (*<-- error message given here*)
begin
end.
```

In the example, BigSet really only has 256 elements, but is still illegal.

```
program Solve;
type
  BigSet = set of 0..255;
begin
end.
```

We need to make sure the upper and lower bounds and in the range 0..255.

3.1.2.1.438 E2282: Property setters cannot take var parameters

This message is displayed when you try to use a var parameter in a property setter parameter. The parameter of a property setter procedure cannot be a var or out parameter.

3.1.2.1.439 E2193: Slice standard function only allowed as open array argument

An attempt has been made to pass an array slice to a fixed size array. Array slices can only be sent to open array parameters. none

```
program Produce;

  type
    IntegerArray = array [1..10] OF Integer;

  var
    SliceMe : array [1..200] OF Integer;

  procedure TakesArray(x : IntegerArray);
  begin
  end;

begin TakesArray(SLICE(SliceMe, 5));
end.
```

In the above example, the error is produced because TakesArray expects a fixed size array.

```
program Solve;

  type
    IntegerArray = array [1..10] OF Integer;

  var
    SliceMe : array [1..200] OF Integer;

  procedure TakesArray(x : array of Integer);
  begin
  end;

begin TakesArray(SLICE(SliceMe, 5));
end.
```

In the above example, the error is not produced because TakesArray takes an open array as the parameter.

3.1.2.1.440 E2454: Slice standard function not allowed for VAR nor OUT argument

You cannot write back to a slice of an array, so you cannot use the slice standard function to pass an argument that is **var** or **out**. If you must modify the array, either pass in the full array or use an array variable to hold the desired part of the full array.

3.1.2.1.441 E2240: $EXTERNALSYM and $NODEFINE not allowed for '%s'; only global symbols

The $EXTERNALSYM and $NODEFINE directives can only be applied to global symbols.

3.1.2.1.442 W1014: String constant truncated to fit STRING[%ld]

A string constant is being assigned to a variable which is not large enough to contain the entire string. The compiler is alerting you to the fact that it is truncating the literal to fit into the variable. -W

```
program Produce;
```

```
(*$WARNINGS ON*)

  const
    Title = 'Super Galactic Invaders with Turbo Gungla Sticks';
    Subtitle = 'Copyright (c) 2002 by Frank Borland';

  type
    TitleString = String[25];
    SubtitleString = String[18];


  var
    ProgramTitle : TitleString;
    ProgramSubtitle : SubtitleString;

begin
  ProgramTitle := Title;
  ProgramSubtitle := Subtitle;
end.
```

The two string constants are assigned to variables which are too short to contain the entire string. The compiler will truncate the strings and perform the assignment.

```
program Solve;
(*$WARNINGS ON*)

  const
    Title = 'Super Galactic Invaders with Turbo Gungla Sticks';
    Subtitle = 'Copyright (c) 2002';

  type
    TitleString = String[55];
    SubtitleString = String[18];


  var
    ProgramTitle : TitleString;
    ProgramSubtitle : SubtitleString;

begin
  ProgramTitle := Title;
  ProgramSubtitle := Subtitle;
end.
```

There are two solutions to this problem, both of which are demonstrated in this example. The first solution is to increase the size of the variable to hold the string. The second is to reduce the size of the string to fit in the declared size of the variable.

3.1.2.1.443 E2354: String element cannot be passed to var parameter

No further information is available for this error or warning.

3.1.2.1.444 E2056: String literals may have at most 255 elements

This error message occurs when you declare a string type with more than 255 elements, if you assign a string literal of more than 255 characters to a variable of type ShortString, or when you have more than 255 characters in a single character string.

Note that you can construct long string literals spanning more than one line by using the '+' operator to concatenate several string literals.

```
program Produce;
var
  LongString : string[256];  (*<-- Error message here*)
begin
```

```
end.
```

In the example above, the length of the string is just one beyond the limit.

```
program Solve;
var
  LongString : AnsiString;
begin
end.
```

The most convenient solution is to use the new long strings - then you don't even have to spend any time thinking about what a reasonable maximum length would be.

3.1.2.1.445 E2408: Can't extract strong name key from assembly %s

No further information is available for this error or warning.

3.1.2.1.446 W1044: Suspicious typecast of %s to %s

This warning flags typecasts like PWideChar(String) or PChar(WideString) which are casting between different string types without character conversion.

3.1.2.1.447 E2272: Cannot use reserved unit name '%s'

An attempt has been made to use one of the reserved unit names, such as System, as the name of a user-created unit.

The names in the following list are currently reserved by the compiler.

- System
- SysInit

```
unit System;
interface
implementation
begin
end.
```

The name of the unit in this example is illegal because it is reserved for use by the compiler.

```
unit MySystem;
interface
implementation
begin
end.
```

The only solution to this problem is to use a different name for the unit.

3.1.2.1.448 E2156: Expression too complicated

The compiler has encounter an expression in your source code that is too complicated for it to handle.

Reduce the complexity of your expression by introducing some temporary variables.

3.1.2.1.449 E2283: Too many local constants. Use shorter procedures

One or more of your procedures contain so many string constant expressions that they exceed the compiler's internal storage limit. This can occur in code that is automatically generated. To fix this, you can shorten your procedures or declare contant identifiers instead of using so many literals in the code.

3.1.2.1.450 E2163: Too many conditional symbols

You have exceeded the memory allocated to conditional symbols defined on the command line (including configuration files). There are 256 bytes allocated for all the conditional symbols. Each conditional symbol requires 1 extra byte when stored in conditional symbol area.

The only solution is to reduce the number of conditional compilation symbols contained on the command line (or in configuration files).

3.1.2.1.451 E2226: Compilation terminated; too many errors

The compiler has surpassed the maximum number of errors which can occur in a single compilation.

The only solution is to address some of the errors and recompile the project.

3.1.2.1.452 E2034: Too many actual parameters

This error message occurs when a procedure or function call gives more parameters than the procedure or function declaration specifies.

Additionally, this error message occurs when an OLE automation call has too many (more than 255), or too many named parameters.

```
program Produce;

function Max(A,B: Integer): Integer;
begin
  if A > B then Max := A else Max := B
end;

begin
  Writeln( Max(1,2,3) );   (*<-- Error message here*)
end.
```

It would have been convenient for Max to accept three parameters...

```
program Solve;

function Max(const A: array of Integer): Integer;
var
  I: Integer;
begin
  Result := Low(Integer);
  for I := 0 to High(A) do
    if Result < A[I] then
      Result := A[I];
end;

begin
  Writeln( Max([1,2,3]) );
end.
```

Normally, you would change to call site to supply the right number of parameters. Here, we have chose to show you how to implement Max with an unlimited number of arguments. Note that now you have to call it in a slightly different way.

3.1.2.1.453 E2436: Type declarations not allowed in anonymous record or local record type

Record types that are declared in local scopes or declared in-place in variable declarations can only contain field declarations. For advanced features in record types (such as methods, properties, and nested types), the record type must be an explicitly declared global type.

3.1.2.1.454 E2005: '%s' is not a type identifier

This error message occurs when the compiler expected the name of a type, but the name it found did not stand for a type.

```
program Produce;
type
  TMyClass = class
    Field: Integer;
  end;
var
  MyClass: TMyClass;

procedure Proc(C: MyClass);           (*<-- Error message here*)
begin
end;

begin
end.
```

The example erroneously uses the name of the variable, not the name of the type, as the type of the argument.

```
program Solve;
type
  TMyClass = class
    Field: Integer;
  end;
var
  MyClass: TMyClass;

procedure Proc(C: TMyClass);
begin
end;

begin
end.
```

Make sure the offending identifier is indeed a type - maybe it was misspelled, or another identifier of the same name hides the one you meant to refer to.

3.1.2.1.455 x2243: Expression needs no Initialize/Finalize

You have attempted to use the standard procedure Finalize on a type that requires no finalization.

```
program Produce;

  var
    ch : Char;

begin
  Finalize(ch);
end.
```

In this example, the Delphi type Char needs no finalization.

The usual solution to this problem is to remove the offending use of Finalize.

3.1.2.1.456 E2100: Data type too large: exceeds 2 GB

You have specified a data type which is too large for the compiler to represent. The compiler will generate this error for datatypes which are greater or equal to 2 GB in size. You must decrease the size of the description of the type.

```
program Produce;

  type
    EnormousArray = array [0..MaxLongint] OF Longint;
    BigRecord = record
      points : array [1..10000] of Extended;
    end;

  var
    data : array [0..500000] of BigRecord;

begin
end.
```

It is easily apparent to see why these declarations will elicit error messages.

```
program Solve;
  type
    EnormousArray = array [0..MaxLongint DIV 8] OF Longint;

    DataPoints = ^DataPointDesc;
    DataPointDesc = array [1..10000] of Extended;
    BigRecord = record
      points : DataPoints;
    end;

  var
    data : array [0..500000] OF BigRecord;


begin
end.
```

The easy solution to avoid this error message is to make sure that the size of your data types remain under 2Gb in size. A more complicated method would involve the restructuring of your data, as has been begun with the BigRecord declaration.

3.1.2.1.457 E2101: Size of data type is zero

Record types must contain at least one instance data field. Zero-size records are not allowed in .NET.

3.1.2.1.458 W1016: Typed constant '%s' passed as var parameter

This error message is reserved.

3

3.1.2.1.459 W1055: Published caused RTTI ($M+) to be added to type '%s'

You added a 'PUBLISHED' section to a class that was not compiled while the {$M+}/{$TYPEINFO ON} switch was in effect, or without deriving from a class compiled with the {$M+}/{$TYPEINFO ON} switch in effect.

The TypeInfo standard procedure requires a type identifier as its parameter. In the code above, 'NotType' does not represent a type identifier.

To avoid this error, ensure that you compile while the {$M+}/{$TYPEINFO ON} switch is on, or derive from a class that was compiled with {$M+}/{$TYPEINFO ON} switch on.

3.1.2.1.460 E2133: TYPEINFO standard function expects a type identifier

You have attempted to obtain type information for an identifier which does not represent a type.

```
program Produce;

  var
    p : Pointer;

  procedure NotType;
  begin
  end;


begin
  p := TypeInfo(NotType);
end.
```

The TypeInfo standard procedure requires a type identifier as it's parameter. In the code above, 'NotType' does not represent a type identifier.

```
program Solve;

  type
    Base = class
    end;

  var
    p : Pointer;

begin
  p := TypeInfo(Base);
end.
```

By ensuring that the parameter used for TypeInfo is a type identifier, you will avoid this error.

3.1.2.1.461 E2147: Property '%s' does not exist in base class

The compiler believes you are attempting to hoist a property to a different visibility level in a derived class, but the specified property does not exist in the base class.

```
program Produce;

  type
    Base = class
    private
      a : Integer;
      property BaseProp : integer read a write a;
    end;

    Derived = class (Base)
      ch : Char;
      property Alpha read ch write ch; (*case 1*)
      property BesaProp; (*case 2*)
    end;

begin
end.
```

There are two basic causes of this error. The first is the specification of a new property without specifying a type; this usually is not supposed to be a movement to a new visibility level. The second is the specification of a property which should exist in the base class, but is not found by the compiler; the most likely cause for this is a simple typo (as in "BesaProp"). In the second form, the compiler will also output errors that a read or write clause was expected.

```
program Solve;

  type
    Base = class
    private
      a : Integer;
      property BaseProp : integer read a write a;
    end;

    Derived = class (Base)
      ch : Char;
    public
      property Alpha : Char read ch write ch; (*case 1*)
      property BaseProp; (*case 2*)
    end;

begin
end.
```

The solution for the first case is to supply the type of the property. The solution for the second case is to check the spelling of the property name.

3.1.2.1.462 E2452: Unicode characters not allowed in published symbols

The VCL Run-Time Type Information (RTTI) subsystem and the streaming of DFM files require that published symbols are non-Unicode (ANSI) characters. Consider whether this symbol needs to be published, and if so, use ANSI characters instead of Unicode.

3.1.2.1.463 W1041: Error converting Unicode char to locale charset. String truncated. Is your LANG environment variable set correctly?

This message occurs when you are trying to convert strings in Unicode to your local character set and the string contains characters that are not valid for the current locale. For example, this may occur when converting WideString to AnsiString or if attempting to display Japanese characters in an English locale.

3.1.2.1.464 W1006: Unit '%s' is deprecated

The unit is deprecated, but continues to be available to support backward compatibility.

The unit is tagged (using the **deprecated** hint directive) as no longer current and is maintained for compatibility only. You should consider updating your source code to use another unit, if possible.

The **$WARN** UNIT_DEPRECATED ON/OFF compiler directive turns on or off all warnings about the **deprecated** directive in units where the **deprecated** directive is specified.

3.1.2.1.465 W1007: Unit '%s' is experimental

An "experimental" directive has been used on an identifier. "Experimental" indicates the presence of a class or unit which is incomplete or not fully tested.

3.1.2.1.466 F2048: Bad unit format: '%s'

This error occurs when a compiled unit file (.dcu file) has a bad format.

Most likely, the file has been corrupted. Recompile the file if you have the source. If the problem persists, you may have to reinstall Delphi.

3.1.2.1.467 W1052: Can't find System.Runtime.CompilerServices.RunClassConstructor. Unit initialization order will not follow uses clause order

This warning indicates that the initialization order defined by the Delphi language, that specified by the order of units in the uses clause, is not guaranteed.

The RunClassConstructor function is used to execute the initialization sections of units used by the current unit in the order specified by the current unit's **uses** clauses. This warning will be issued if the compiler cannot find this function in the .NET Framework you are linking against. For example, it will occur when linking against the .NET Compact Framework, which does not implement RunClassConstructor.

3.1.2.1.468 W1004: Unit '%s' is specific to a library

The whole unit is tagged (using the **library** hint directive) as one that may not be available in all libraries. If you are likely to use different libraries, it may cause a problem.

The **$WARN** UNIT_LIBRARY ON/OFF compiler directive turns on or off all warnings in units where the **library** directive is specified.

3.1.2.1.469 E1038: Unit identifier '%s' does not match file name

The unit name in the top unit is case sensitive and must match the name with respect to upper- and lowercase letters exactly. The unit name is case sensitive only in the unit declaration.

3.1.2.1.470 W1005: Unit '%s' is specific to a platform

The whole unit is tagged (using the **platform** hint directive) as one that contains material that may not be available on all platforms. If you are writing cross-platform applications, it may cause a problem. For example, a unit that uses objects defined in OleAuto might be tagged using the PLATFORM directive

The **$WARN** UNIT_PLATFORM ON/OFF compiler directive turns on or off all warnings about the **platform** directive in units where the **platform** directive is specified.

3.1.2.1.471 E2070: Unknown directive: '%s'

This error message appears when the compiler encounters an unknown directive in a procedure or function declaration.

The directive is probably misspelled, or a semicolon is missing.

```
program Produce;

procedure P; stcall;
begin
```

```
end;

procedure Q forward;

function GetLastError: Integer external 'kernel32.dll';

begin
end.
```

In the declaration of P, the calling convention "stdcall" is misspelled. In the declaration of Q and GetLastError, we're missing a semicolon.

```
program Solve;

procedure P; stdcall;
begin
end;

procedure Q; forward;

function GetLastError: Integer; external 'kernel32.dll';

begin
end.
```

The solution is to make sure the directives are spelled correctly, and that the necessary semicolons are there.

3.1.2.1.472 E2328: Linker error while emitting metadata

No further information is available for this error or warning.

3.1.2.1.473 E2400: Unknown Resource Format '%s'

No further information is available for this error or warning.

3.1.2.1.474 E2216: Can't handle section '%s' in object file '%s'

You are trying to link object modules into your program with the $L compiler directive. However, the object file is too complex for the compiler to handle. For example, you may be trying to link in a C++ object file. This is not supported.

3.1.2.1.475 E2405: Unknown element type found importing signature of %s.%s

No further information is available for this error or warning.

3.1.2.1.476 E2417: Field offset cannot be determined for variant record because previous field type is unknown size record type

Private types in an assembly are not imported and are marked as having an unreliable size. If a record is declared as having at least one private field or it has one field whose type size is unreliable then this error will occur.

3.1.2.1.477 E2166: Unnamed arguments must precede named arguments in OLE Automation call

You have attempted to follow named OLE Automation arguments with unnamed arguments.

```
program Produce;

  var
    ole : variant;

begin ole.dispatch(filename:='FrogEggs', 'Tapioca');
end.
```

The named argument, 'filename' must follow the unnamed argument in this OLE dispatch.

```
program Solve;

  var
    ole : variant;

begin ole.dispatch('Tapioca', filename:='FrogEggs');
end.
```

This solution, reversing the parameters, is the most straightforward but it may not be appropriate for your situation. Another alternative would be to provide the unnamed parameter with a name.

3.1.2.1.478 E2289: Unresolved custom attribute: %s

A custom attribute declaration was not followed by a symbol declaration such as a type, variable, method, or parameter declaration.

3.1.2.1.479 W1048: Unsafe typecast of '%s' to '%s'

You have used a data type or operation for which static code analysis cannot prove that it does not overwrite memory. In a secured execution environment such as .NET, such code is assumed to be unsafe and a potential security risk.

3.1.2.1.480 W1047: Unsafe code '%s'

You have used a data type or operation for which static code analysis cannot prove that it does not overwrite memory. In a secured execution environment such as .NET, such code is assumed to be unsafe and a potential security risk.

3.1.2.1.481 E2406: EXPORTS section allowed only if compiling with {$UNSAFECODE ON}

No further information is available for this error or warning.

3.1.2.1.482 W1046: Unsafe type '%s%s%s'

You have used a data type or operation for which static code analysis cannot prove that it does not overwrite memory. In a secured execution environment such as .NET, such code is assumed to be unsafe and a potential security risk.

3.1.2.1.483 E2396: Unsafe code only allowed in unsafe procedure

No further information is available for this error or warning.

3.1.2.1.484 E2395: Unsafe procedure only allowed if compiling with {$UNSAFECODE ON}

No further information is available for this error or warning.

3.1.2.1.485 E2397: Unsafe pointer only allowed if compiling with {$UNSAFECODE ON}

No further information is available for this error or warning.

3.1.2.1.486 E2410: Unsafe pointer variables, parameters or consts only allowed in unsafe procedure

No further information is available for this error or warning.

3.1.2.1.487 x1025: Unsupported language feature: '%s'

You are attempting to translate a Delphi unit to a C++ header file which contains unsupported language features.

You must remove the offending construct from the interface section before the unit can be translated.

3.1.2.1.488 E2057: Unexpected end of file in comment started on line %ld

This error occurs when you open a comment, but do not close it.

Note that a comment started with '{' must be closed with '}', and a comment started with '(*' must be closed with '*)'.

```
program Produce;
(*Let's start a comment here but forget to close it
begin
end.
```

So the example just didn't close the comment.

```
program Solve;
(*Let's start a comment here and not forget to close it*)
begin
end.
```

Doing so fixes the problem.

3.1.2.1.489 E2280: Unterminated conditional directive

For every {$IFxxx}, the corresponding {$ENDIF} or {$IFEND} must be found within the same source file. This message indicates that you do not have an equal number of ending directives.

This error message is reported at the source line of the last $IF/$IFDEF/etc. with no matching $ENDIF/$IFEND. This gives you a good place to start looking for the source of the problem.

3.1.2.1.490 E2052: Unterminated string

The compiler did not find a closing apostrophe at the end of a character string.

Note that character strings cannot be continued onto the next line - however, you can use the '+' operator to concatenate two character strings on separate lines.

```
program Produce;

begin
  Writeln('Hello world!);   (*<-- Error message here -*)
end.
```

We just forgot the closing quote at the string - no big deal, happens all the time.

```
program Solve;

begin
  Writeln('Hello world!');
end.
```

So we supplied the closing quote, and the compiler is happy.

3.1.2.1.491 H2164: Variable '%s' is declared but never used in '%s'

You have declared a variable in a procedure, but you never actually use it. -H

```
program Produce;
(*$HINTS ON*)

  procedure Local;
    var i : Integer;
  begin
  end;

begin
end.
    program Solve;

(*$HINTS ON*)

  procedure Local;
  begin
  end;

begin
end.
```

One simple solution is to remove any unused variable from your procedures. However, unused variables can also indicate an error in the implementation of your algorithm.

3.1.2.1.492 W1036: Variable '%s' might not have been initialized

This warning is given if a variable has not been assigned a value on every code path leading to a point where it is used.

```
program Produce;
(*$WARNINGS ON*)
var
  B: Boolean;
  C: (Red,Green,Blue);

procedure Simple;
var
  I : Integer;
begin
  Writeln(I);        (*<-- Warning here*)
end;
```

3

```
procedure IfStatement;
var
  I : Integer;
begin
  if B then
    I := 42;
  Writeln(I);        (*<-- Warning here*)
end;

procedure CaseStatement;
var
  I: Integer;
begin
  case C of
  Red..Blue: I := 42;
  end;
  Writeln(I);        (*<-- Warning here*)
end;

procedure TryStatement;
var
  I: Integer;
begin
  try
    I := 42;
  except
    Writeln('Should not get here!');
  end;
  Writeln(I);        (*<-- Warning here*)
end;

begin
  B := False;
end.
```

In an if statement, you have to make sure the variable is assigned in both branches. In a case statement, you need to add an else part to make sure the variable is assigned a value in every conceivable case. In a try-except construct, the compiler assumes that assignments in the try part may in fact not happen, even if they are at the very beginning of the try part and so simple that they cannot conceivably cause an exception.

```
program Solve;
(*$WARNINGS ON*)
var
  B: Boolean;
  C: (Red,Green,Blue);

procedure Simple;
var
  I : Integer;
begin
  I := 42;
  Writeln(I);
end;

procedure IfStatement;
var
  I : Integer;
begin
  if B then
    I := 42
  else
    I := 0;
  Writeln(I);        (*Need to assign I in the else part
end;

procedure CaseStatement;
```

```
var
  I: Integer;
begin
  case C of
  Red..Blue: I := 42;
  else       I := 0;
  end;
  Writeln(I);        (*Need to assign I in the else part*)
end;

procedure TryStatement;
var
  I: Integer;
begin
  I := 0;
  try
    I := 42;
  except
    Writeln('Should not get here!');
  end;
  Writeln(I);        (*Need to assign I before the try*)
end;

begin
  B := False;
end.
```

The solution is to either add assignments to the code paths where they were missing, or to add an assignment before a conditional statement or a try-except construct.

3.1.2.1.493 E2157: Element 0 inaccessible - use 'Length' or 'SetLength'

The Delphi String type does not store the length of the string in element 0. The old method of changing, or getting, the length of a string by accessing element 0 does not work with long strings.

```
program Produce;

  var
    str : String;
    len : Integer;

begin
  str := 'Kojo no tsuki';
  len := str[0];
end.
```

Here the program is attempting to get the length of the string by directly accessing the first element. This is not legal.

```
program Solve;

  var
    str : String;
    len : Integer;

begin
  str := 'Kojo no tsuki';
  len := Length(str);
end.
```

You can use the SetLength and Length standard procedures to provide the same functionality as directly accessing the first element of the string. If hints are turned on, you will receive a warning about the value of 'len' not being used.

3.1.2.1.494 E2255: New not supported for dynamic arrays - use SetLength

The program has attempted to use the standard procedure NEW on a dynamic array. The proper method for allocating dynamic arrays is to use the standard procedure SetLength.

```
program Produce;
  var
    arr : array of integer;

begin
  new(arr, 10);
end.
```

The standard procedure NEW cannot be used on dynamic arrays.

```
program Solve;
  var
    arr : array of integer;

begin
  SetLength(arr, 10);
end.
```

Use the standard procedure SetLength to allocate dynamic arrays.

3.1.2.1.495 E2212: Package unit '%s' cannot appear in contains or uses clauses

The unit named in the error is a package unit and as such cannot be included in your project. A possible cause of this error is that somehow a Delphi unit and a package unit have been given the same name. The compiler is finding the package unit on its search path before it can locate a same-named Delphi file. Packages cannot be included in a project by inclusion of the package unit in the uses clause.

3.1.2.1.496 F2063: Could not compile used unit '%s'

This fatal error is given when a unit used by another could not be compiled. In this case, the compiler gives up compilation of the dependent unit because it is likely very many errors will be encountered as a consequence.

3.1.2.1.497 E2090: User break - compilation aborted

This message is currently unused.

3.1.2.1.498 E2165: Compile terminated by user

You pressed Ctrl-Break during a compile.

3.1.2.1.499 E2142: Inaccessible value

You have tried to view a value that is not accessible from within the integrated debugger. Certain types of values, such as a 0 length Variant-type string, cannot be viewed within the debugger.

3.1.2.1.500 H2077: Value assigned to '%s' never used

The compiler gives this hint message if the value assigned to a variable is not used. If optimization is enabled, the assignment is eliminated.

This can happen because either the variable is not used anymore, or because it is reassigned before it is used.

```
program Produce;
(*$HINTS ON*)

procedure Simple;
var
  I: Integer;
begin
  I := 42;                (*<-- Hint message here*)
end;

procedure Propagate;
var
  I: Integer;
  K: Integer;
begin
  I := 0;                 (*<-- Hint message here*)
  Inc(I);                 (*<-- Hint message here*)
  K := 42;
  while K > 0 do begin
    if Odd(K) then
      Inc(I);             (*<-- Hint message here*)
    Dec(K);
  end;
end;

procedure TryFinally;
var
  I: Integer;
begin
  I := 0;                 (*<-- Hint message here*)
  try
    I := 42;
  finally
    Writeln('Reached finally');
  end;
  Writeln(I);             (*Will always write 42 - if an exception happened,
          we wouldn't get here*)
end;

begin
end.
```

In procedure Propagate, the compiler is smart enough to realize that as variable I is not used after the while loop, it does not need to be incremented inside the while, and therefore the increment and the assignment before the while loop are also superfluous.

In procedure TryFinally, the assignment to I before the try-finally construct is not necessary. If an exception happens, we don't execute the Writeln statement at the end, so the value of I does not matter. If no exception happens, the value of I seen by the Writeln statement is always 42. So the first assignment will not change the behavior of the procedure, and can therefore be eliminated.

This hint message does not indicate your program is wrong - it just means the compiler has determined there is an assignment that is not necessary.

You can usually just delete this assignment - it will be dropped in the compiled code anyway if you compile with optimizations on.

Sometimes, however, the real problem is that you assigned to the wrong variable, for example, you meant to assign J but instead assigned I. So it is worthwhile to check the assignment in question carefully.

3.1.2.1.501 E2088: Variable name expected

This error message is issued if you try to declare an absolute variable, but the absolute directive is not followed by an integer constant or a variable name.

```
program Produce;

var
  I : Integer;
  J : Integer absolute Addr(I);   (*<-- Error message here*)

begin
end.
    program Solve;

const
  Addr = 0;

var
  I : Integer;
  J : Integer absolute I;

begin
end.
```

3.1.2.1.502 E2171: Variable '%s' inaccessible here due to optimization

The evaluator or watch statement is attempting to retrieve the value of <name>, but the compiler was able to determine that the variables actual lifetime ended prior to this inspection point. This error will often occur if the compiler determines a local variable is assigned a value that is not used beyond a specific point in the program's control flow.

```
Create a new application.
Place a button on the form.
Double click the button to be taken to the 'click' method.
Add a global variable, 'c', of type Integer to the implementation section.

The click method should read as:

  procedure TForm1.Button1Click(Sender: TObject);
    var a, b : integer;
  begin
    a := 10;
    b := 20;
    c := b;
    a := c;
  end;

Set a breakpoint on the assignment to 'c'.
Compile and run the application.
Press the button.
After the breakpoint is reached, open the evaluator (Run|Evaluate/Watch).
Evaluate 'a'.
```

The compiler realizes that the first assignment to 'a' is dead, since the value is never used. As such, it defers even using 'a' until the second assignment occurs - up until the point where 'c' is assigned to 'a', the variable 'a' is considered to be dead and cannot be used by the evaluator.

The only solution is to only attempt to view variables which are known to have live values.

3.1.2.1.503 E2033: Types of actual and formal var parameters must be identical

For a variable parameter, the actual argument must be of the exact type of the formal parameter.

```
program Produce;

procedure SwapBytes(var B1, B2: Byte);
var
  Temp: Byte;
begin
  Temp := B1; B1 := B2; B2 := Temp;
end;

var
  C1, C2: 0..255;     (*Similar to a byte, but NOT identical*)
begin
  SwapBytes(C1,C2);   (*<-- Error message here*)
end.
```

Arguments C1 and C2 are not acceptable to SwapBytes, although they have the exact memory representation and range that a Byte has.

```
program Solve;

procedure SwapBytes(var B1, B2: Byte);
var
  Temp: Byte;
begin
  Temp := B1; B1 := B2; B2 := Temp;
end;

var
  C1, C2: Byte;
begin
  SwapBytes(C1,C2);   (*<-- No error message here*)
end.
```

So you actually have to declare C1 and C2 as Bytes to make this example compile.

3.1.2.1.504 E2277: Only external cdecl functions may use varargs

This message indicates that you are trying to implement a varargs routine. You cannot implement varargs routines, you can only call external varargs.

3.1.2.1.505 F2051: Unit %s was compiled with a different version of %s.%s

This fatal error occurs when the declaration of symbol declared in the interface part of a unit has changed, and the compiler cannot recompile a unit that relies on this declaration because the source is not available to it.

There are several possible solutions - recompile Unit1 (assuming you have the source code available), use an older version of Unit2 or change Unit2, or get a new version of Unit1 from whoever has the source code.

This error can also occur when a unit in your project has the same name as a standard Delphi unit.

For example, this may occur is when compiling a project written in a previous version of Delphi that did not have a unit of this name (for example, search.pas was not in Delphi 3).

To solve the problem in this case:

1. Open <Unit2> and save it with a new name.

2. Alter all references to <Unit2> in uses clauses to refer to the new name.
3. Delete the old <Unit2>.pas AND <Unit2>.dcu versions of this unit.
4. Rebuild the project.

3.1.2.1.506 E2379: Virtual methods not allowed in record types

No further information is available for this error or warning.

3.1.2.1.507 E2423: Void type not usable in this context

The System type Void is not allowed to be used in some contexts. As an example, the following code demostrates the contexts where type Void may not be used.

```
program Project3;

{$APPTYPE CONSOLE}

type
  TBar = class
    property Bar: Void;

  end;

  TBaz = type Void;

var
  TFoo: ^Void;

procedure Bar(Arg: Void);
begin
end;

function Foo: Void;
begin
end;

end.
```

3.1.2.1.508 E2221: $WEAKPACKAGEUNIT '%s' cannot have initialization or finalization code

A unit which has been flagged with the $weakpackageunit directive cannot contain initialization or finalization code, nor can it contain global data. The reason for this is that multiple copies of the same weakly packaged units can appear in an application, and then referring to the data for that unit becomes and ambiguous proposition. This ambiguity is furthered when dynamically loaded packages are used in your applications.

```
(*$WEAKPACKAGEUNIT*)
unit yamadama;
interface
implementation
  var
    Title : String;

initialization
  Title := 'Tiny Calc';
finalization
end.
```

In the above example, there are two problems: Title is a global variable, and Title is initialized in the initialization section of the unit.

There are only two alternatives: either remove the $weakpackageunit directive from the unit, or remove all global data, initialization and finalization code.

3.1.2.1.509 E2203: $WEAKPACKAGEUNIT '%s' contains global data

A unit which was marked with $WEAKPACKAGEUNIT is being placed into a package, but it contains global data. It is not legal for such a unit to contain global data or initialization or finalization code.

The only solutions to this problem are to remove the $WEAKPACKAGEUNIT mark, or remove the global data from the unit before it is put into the package.

3.1.2.1.510 W1050: WideChar reduced to byte char in set expressions

"Set of char" in Win32 defines a set over the entire range of the Char type. Since Char is a byte-sized type in Win32, this defines a set of maximum size containing 256 elements. In .NET, Char is a word-sized type, and this range (0..65535) exceeds the capacity of the set type.

To accomodate existing code that uses this "Set of Char" syntax, the compiler will treat the expression as "set of AnsiChar". The warning message reminds you that the set can only store the boolean state of 256 distinct elements, not the full range of the Char type.

3.1.2.1.511 E2152: Wrong or corrupted version of RLINK32.DLL

The internal consistency check performed on the RLINK32.DLL file has failed.

Contact CodeGear if you encounter this error.

3.1.2.1.512 E2015: Operator not applicable to this operand type

This error message is given whenever an operator cannot be applied to the operands it was given - for instance if a boolean operator is applied to a pointer.

```
program Produce;
var
  P: ^Integer;
begin
  if P and P^ > 0 then
    Writeln('P points to a number greater 0');
end.
```

Here a C++ programmer was unclear about operator precedence in Delphi - P is not a boolean expression, and the comparison needs to be parenthesized.

```
program Solve;
var
  P: ^Integer;
begin
  if (P <> nil) and (P^ > 0) then
    Writeln('P points to a number greater 0');
end.
```

If we explicitly compare P to nil and use parentheses, the compiler is happy.

3.1.2.1.513 W1206: XML comment on '%s' has cref attribute '%s' that could not be resolved

This warning message occurs when the XML has a cref attribute that cannot be resolved.

This is a warning in XML documentation processing. The XML is well formed, but the comment's meaning is questionable. XML cref references follow the .NET style. See http://msdn2.microsoft.com/en-us/library/acd0tfbe.aspx for more details. A documentation warning does not prevent building.

3.1.2.1.514 W1205: XML comment on '%s' has badly formed XML--'The character '%c' was expected.'

This warning message occurs when the expected character was not found in the XML.

This is an error in XML documentation processing. The XML is not well formed. This is a warning because a documentation error does not prevent building.

3.1.2.1.515 W1204: XML comment on '%s' has badly formed XML--'A name contained an invalid character.'

This warning message occurs when a name in XML contains an invalid character.

This is an error in XML documentation processing. The XML is not well formed. This is a warning because a documentation error does not prevent building.

3.1.2.1.516 W1203: XML comment on '%s' has badly formed XML--'A name was started with an invalid character.'

This warning message occurs when an XML name was started with an invalid character.

This is an error in XML documentation processing. The XML is not well formed. This is a warning because a documentation error does not prevent building.

3.1.2.1.517 W1208: Parameter '%s' has no matching param tag in the XML comment for '%s' (but other parameters do)

This warning message occurs when an XML Parameter has no matching param tag in the XML comment but other parameters do.

This is a warning in XML documentation processing. There is at least one tag, but some parameters in the method don't have a tag. A documentation warning does not prevent building.

3.1.2.1.518 W1207: XML comment on '%s' has a param tag for '%s', but there is no parameter by that name

This warning message occurs when the XML contains a parameter tag for a nonexistent parameter.

This is a warning in XML documentation processing. The XML is well formed, however, a tag was created for a parameter that doesn't exist in a method. A documentation warning does not prevent building.

3.1.2.1.519 W1202: XML comment on '%s' has badly formed XML--'Reference to undefined entity '%s''

This warning message occurs when XML references an undefined entity.

This is an error in XML documentation processing. The XML is not well formed. This is a warning because a documentation error does not prevent building.

3.1.2.1.520 W1201: XML comment on '%s' has badly formed XML--'Whitespace is not allowed at this location.'

This warning message occurs when the compiler encounters white space in a location in which white space is not allowed.

This is an error in XML documentation processing. The XML is not well formed. This is a warning because a documentation error does not prevent building.

3.1.2.1.521 W1013: Constant 0 converted to NIL

The Delphi compiler now allows the constant 0 to be used in pointer expressions in place of NIL. This change was made to allow older code to still compile with changes which were made in the low-level RTL.

```
program Produce;

  procedure p0(p : Pointer);
  begin
  end;

begin
  p0(0);
end.
```

In this example, the procedure p0 is declared to take a Pointer parameter yet the constant 0 is passed. The compiler will perform the necessary conversions internally, changing 0 into NIL, so that the code will function properly.

```
program Solve;

  procedure p0(p : Pointer);
  begin
  end;

begin
  p0(NIL);
end.
```

There are two approaches to solving this problem. In the case above the constant 0 has been replaced with NIL. Alternatively the procedure definition could be changed so that the parameter type is of Integer type.

3.1.2.2 Delphi Runtime Errors

Certain errors at runtime cause Delphi programs to display an error message and terminate.

Identifying Runtime Errors

Runtime errors take the form:

```
Runtime error nnn at xxxxxxxx
```

where nnn is the runtime error number, and xxxxxxxx is the runtime error address.

Applications that use the SysUtils class map most runtime errors to Exceptions, which allow your application to resolve the error without terminating.

Types of Runtime Errors

Delphi runtime errors are divided into the following categories:

- I/O errors, numbered 100 through 149
- Fatal errors, numbered 200 through 255
- Operating system errors

See Also

Exception handling (see page 541)

Resolving internal errors (see page 130)

Fatal errors (see page 511)

I/O errors (see page 510)

Operating system errors (see page 512)

3.1.2.3 I/O Errors

I/O errors cause an exception to be thrown if a statement is compiled in the {$I+} state. (If the application does not include the SysUtils class, the exception causes the application to terminate).

Handling I/O Errors

In the {$I-} state, the program continues to execute, and the error is reported by the IOResult function.

I/O Error List

The following table lists all I/O errors, numbers, and descriptions.

Number	Name	Description
100	Disk read error	Reported by Read on a typed file if you attempt to read past the end of the file.
101	Disk write error	Reported by CloseFile, Write, Writeln, or Flush if the disk becomes full.
102	File not assigned	Reported by Reset, Rewrite, Append, Rename, or Erase if the file variable has not been assigned a name through a call to Assign or AssignFile.
103	File not open	Reported by CloseFile, Read Write, Seek, Eof, FilePos, FileSize, Flush, BlockRead, or BlockWrite if the file is not open.
104	File not open for input	Reported by Read, Readln, Eof, Eoln, SeekEof, or SeekEoln on a text file if the file is not open for input.
105	File not open for output	Reported by Write or Writeln on a text file if you do not generate a Console application.
106	Invalid numeric format	Reported by Read or Readln if a numeric value read from a text file does not conform to the proper numeric format.

See Also

Exception handling (see page 541)

Resolving internal errors (see page 130)

Runtime errors (see page 509)

Fatal errors (see page 511)

Operating system errors (see page 512)

3.1.2.4 Fatal errors

These errors always immediately terminate the program.

Exception mapping

In applications that use the SysUtils class (as most GUI applications do), these errors are mapped to exceptions. For a description of the conditions that produce each error, see the documentation for the exception.

I/O error list

The following table lists all fatal errors, numbers, and mapped exceptions.

Number	Name	Exception
200	Division by zero	EDivByZero
201	Range check error	ERangeError
202	Stack overflow	EStackOverflow
203	Heap overflow error	EOutOfMemory
204	Invalid pointer operation	EInvalidPointer
205	Floating point overflow	EOverflow
206	Floating point underflow	EUnderflow
207	Invalid floating point operation	EInvalidOp
210	Abstract Method Error	EAbstractError
215	Arithmetic overflow (integer only)	EIntOverflow
216	Access violation	EAccessViolation
217	Control-C	EControlC
218	Privileged instruction	EPrivilege
219	Invalid typecast	EInvalidCast
220	Invalid variant typecast	EVariantError
221	Invalid variant operation	EVariantError
222	No variant method call dispatcher	EVariantError
223	Cannot create variant array	EVariantError
224	Variant does not contain array	EVariantError
225	Variant array bounds error	EVariantError
226	TLS initialization error	No exception to map to.
227	Assertion failed	EAssertionFailed
228	Interface Cast Error	EIntfCastError
229	Safecall error	ESafecallException
230	Unhandled exception	No exception to map to.
231	Too many nested exceptions	Up to 16 permitted.

232	Fatal signal raised on a non-Delphi thread	No exception to map to.

See Also

Exception handling (see page 541)

Resolving internal errors (see page 130)

Runtime errors (see page 509)

I/O errors (see page 510)

Operating system errors (see page 512)

3.1.2.5 Operating system errors

All errors other than I/O errors and fatal errors are reported with the error codes returned by the operating system.

OS error codes

OS error codes are the return value of operating system function calls. You can obtain the error code for the last error that occurred by calling the global GetLastError function. If you want to raise an exception rather than fetch the error code for the last API call that failed, call the RaiseLastOSError procedure instead.

The error code values returned by GetLastError are dependent on the operating system. You can obtain an error string associated with one of these error codes by calling the global SysErrorMessage function.

Getting return values

To check the return value from a Win32 API function call and raise an EWin32Error exception if it represents an error, call the global Win32Check function.

See Also

Exception handling (see page 541)

Resolving internal errors (see page 130)

Runtime errors (see page 509)

Fatal errors (see page 511)

I/O errors (see page 510)

3.1.3 Delphi Language Guide

The Delphi Language guide describes the Delphi language as it is used in CodeGear development tools. This book describes the Delphi language on both the Win32, and .NET development platforms. Specific differences in the language between the two platforms are marked as appropriate.

Topics

Name	Description
Classes and Objects (see page 513)	This section describes the object-oriented features of the Delphi language, such as the declaration and usage of class types.
Data Types, Variables, and Constants (see page 552)	This section describes the fundamental data types of the Delphi language.
.NET Topics (see page 593)	This section contains information specific to programming in Delphi on the .NET platform.

Generics (Parameterized Types) (↗ see page 595)	Presents an overview of generics, a terminology list, a summary of grammar changes for generics, and details about declaring and using parameterized types, specifying constraints on generics, and using overloads.
Inline Assembly Code (Win32 Only) (↗ see page 609)	This section describes the use of the inline assembler on the Win32 platform.
Object Interfaces (↗ see page 624)	This section describes the use of interfaces in Delphi.
Libraries and Packages (↗ see page 634)	This section describes how to create static and dynamically loadable libraries in Delphi.
Memory Management (↗ see page 644)	This section describes memory management issues related to programming in Delphi on Win32, and on .NET.
Delphi Overview (↗ see page 656)	This chapter provides a brief introduction to Delphi programs, and program organization.
Procedures and Functions (↗ see page 662)	This section describes the syntax of function and procedure declarations.
Program Control (↗ see page 678)	This section describes how parameters are passed to procedures and functions.
Programs and Units (↗ see page 682)	This chapter provides a more detailed look at Delphi program organization.
Standard Routines and I/O (↗ see page 692)	This section describes the standard routines included in the Delphi runtime library.
Fundamental Syntactic Elements (↗ see page 700)	This section describes the fundamental syntactic elements, or the building blocks of the Delphi language.

3.1.3.1 Classes and Objects

This section describes the object-oriented features of the Delphi language, such as the declaration and usage of class types.

Topics

Name	Description
Classes and Objects (↗ see page 514)	This topic covers the following material: • Declaration syntax of classes • Inheritance and scope • Visibility of class members • Forward declarations and mutually dependent classes
Fields (↗ see page 519)	This topic describes the syntax of class data fields declarations.
Methods (↗ see page 521)	A method is a procedure or function associated with a class. A call to a method specifies the object (or, if it is a class method, the class) that the method should operate on. For example, `SomeObject.Free` calls the `Free` method in `SomeObject`. This topic covers the following material: • Methods declarations and implementation • Method binding • Overloading methods • Constructors and destructors • Message methods
Properties (↗ see page 530)	This topic describes the following material: • Property access • Array properties • Index specifiers • Storage specifiers • Property overrides and redeclarations • Class properties

Events (↗ see page 536)	This topic describes the following material: • Event properties and event handlers • Triggering multiple event handlers • Multicast events (.NET)
Class References (↗ see page 539)	Sometimes operations are performed on a class itself, rather than on instances of a class (that is, objects). This happens, for example, when you call a constructor method using a class reference. You can always refer to a specific class using its name, but at times it is necessary to declare variables or parameters that take classes as values, and in these situations you need *class-reference types*. This topic covers the following material: • Class reference types • Class operators
Exceptions (↗ see page 541)	This topic covers the following material: • A conceptual overview of exceptions and exception handling • Declaring exception types • Raising and handling exceptions
Nested Type Declarations (↗ see page 546)	Type declarations can be nested within class declarations. Nested types are used throughout the .NET framework, and throughout object-oriented programming in general. They present a way to keep conceptually related types together, and to avoid name collisions. The same syntax for declaring nested types may be used with the Win32 Delphi compiler.
Operator Overloading (↗ see page 548)	This topic describes Delphi's operator methods and how to overload them.
Class Helpers (↗ see page 550)	This topic describes the syntax of class helper declarations.

3.1.3.1.1 Classes and Objects

This topic covers the following material:

- Declaration syntax of classes
- Inheritance and scope
- Visibility of class members
- Forward declarations and mutually dependent classes

Class Types

A class, or class type, defines a structure consisting of fields, methods, and properties. Instances of a class type are called objects. The fields, methods, and properties of a class are called its components or members.

- A field is essentially a variable that is part of an object. Like the fields of a record, a class' fields represent data items that exist in each instance of the class.
- A method is a procedure or function associated with a class. Most methods operate on objects, that is, instances of a class. Some methods (called class methods) operate on class types themselves.
- A property is an interface to data associated with an object (often stored in a field). Properties have access specifiers, which determine how their data is read and modified. From other parts of a program outside of the object itself a property appears in most respects like a field.

Objects are dynamically allocated blocks of memory whose structure is determined by their class type. Each object has a unique copy of every field defined in the class, but all instances of a class share the same methods. Objects are created and destroyed by special methods called constructors and destructors.

A variable of a class type is actually a pointer that references an object. Hence more than one variable can refer to the same object. Like other pointers, class-type variables can hold the value **nil**. But you don't have to explicitly dereference a class-type variable to access the object it points to. For example, `SomeObject.Size := 100` assigns the value 100 to the

`Size` property of the object referenced by `SomeObject`; you would not write this as `SomeObject^.Size := 100`.

A class type must be declared and given a name before it can be instantiated. (You cannot define a class type within a variable declaration.) Declare classes only in the outermost scope of a program or unit, not in a procedure or function declaration.

A class type declaration has the form

```
type
   className = class [abstract | sealed] (ancestorClass)
      memberList
   end;
```

where *className* is any valid identifier, the **sealed** or **abstract** keyword is optional, *(ancestorClass)* is optional, and *memberList* declares members - that is, fields, methods, and properties - of the class. If you omit *(ancestorClass)*, then the new class inherits directly from the predefined TObject class. If you include *(ancestorClass)* and *memberList* is empty, you can omit **end**. A class type declaration can also include a list of interfaces implemented by the class; see Implementing Interfaces (see page 627).

If a class is marked **sealed**, then it cannot be extended through inheritance. If a class is marked **abstract**, then it cannot be instantiated directly using the `Create` constructor. An entire class can be declared **abstract** even if it does not contain any abstract virtual methods (see page 521). A class cannot be both **abstract** and **sealed**.

Methods appear in a class declaration as function or procedure headings, with no body. Defining declarations for each method occur elsewhere in the program.

For example, here is the declaration of the TMemoryStream class from the `Classes` unit.

```
type TMemoryStream = class(TCustomMemoryStream)
       private
         FCapacity: Longint;
         procedure SetCapacity(NewCapacity: Longint);
       protected
         function Realloc(var NewCapacity: Longint): Pointer; virtual;
         property Capacity: Longint read FCapacity write SetCapacity;
       public
         destructor Destroy; override;
         procedure Clear;
         procedure LoadFromStream(Stream: TStream);
         procedure LoadFromFile(const FileName: string);
         procedure SetSize(NewSize: Longint); override;
         function Write(const Buffer; Count: Longint): Longint; override;
     end;
```

TMemoryStream descends from TCustomMemoryStream (in the `Classes` unit), inheriting most of its members. But it defines - or redefines - several methods and properties, including its destructor method, `Destroy`. Its constructor, `Create`, is inherited without change from TObject, and so is not redeclared. Each member is declared as **private**, **protected**, or **public** (this class has no **published** members). These terms are explained below.

Given this declaration, you can create an instance of TMemoryStream as follows:

```
var stream: TMemoryStream;
    stream := TMemoryStream.Create;
```

Inheritance and Scope

When you declare a class, you can specify its immediate ancestor. For example,

```
type TSomeControl = class(TControl);
```

declares a class called `TSomeControl` that descends from TControl. A class type automatically inherits all of the members from its immediate ancestor. Each class can declare new members and can redefine inherited ones, but a class cannot remove members defined in an ancestor. Hence `TSomeControl` contains all of the members defined in TControl and in each of TControl's ancestors.

The scope of a member's identifier starts at the point where the member is declared, continues to the end of the class declaration, and extends over all descendants of the class and the blocks of all methods defined in the class and its descendants.

TObject and TClass

The TObject class, declared in the `System` unit, is the ultimate ancestor of all other classes. TObject defines only a handful of methods, including a basic constructor and destructor. In addition to TObject, the `System` unit declares the class reference (see page 539) type TClass:

```
TClass = class of TObject;
```

If the declaration of a class type doesn't specify an ancestor, the class inherits directly from TObject. Thus

```
type TMyClass = class
       ...
     end;
```

is equivalent to

```
type TMyClass = class(TObject)
       ...
     end;
```

The latter form is recommended for readability.

Compatibility of Class Types

A class type is assignment-compatible with its ancestors. Hence a variable of a class type can reference an instance of any descendant type. For example, given the declarations

```
type
     TFigure = class(TObject);
     TRectangle = class(TFigure);
     TSquare = class(TRectangle);
var
     Fig: TFigure;
```

the variable `Fig` can be assigned values of type `TFigure`, `TRectangle`, and `TSquare`.

Object Types

The Win32 Delphi compiler allows an alternative syntax to class types, which you can declare object types using the syntax

```
type objectTypeName = object (ancestorObjectType)
        memberList
     end;
```

where *objectTypeName* is any valid identifier, *(ancestorObjectType)* is optional, and *memberList* declares fields, methods, and properties. If *(ancestorObjectType)* is omitted, then the new type has no ancestor. Object types cannot have published members.

Since object types do not descend from TObject, they provide no built-in constructors, destructors, or other methods. You can create instances of an object type using the `New` procedure and destroy them with the `Dispose` procedure, or you can simply declare variables of an object type, just as you would with records.

Object types are supported for backward compatibility only. Their use is not recommended on Win32, and they have been completely deprecated in the Delphi for .NET compiler.

Visibility of Class Members

Every member of a class has an attribute called visibility, which is indicated by one of the reserved words **private**, **protected**, **public**, **published**, or **automated**. For example,

```
published property Color: TColor read GetColor write SetColor;
```

declares a published property called `Color`. Visibility determines where and how a member can be accessed, with **private** representing the least accessibility, **protected** representing an intermediate level of accessibility, and **public**, **published**, and automated representing the greatest accessibility.

If a member's declaration appears without its own visibility specifier, the member has the same visibility as the one that precedes it. Members at the beginning of a class declaration that don't have a specified visibility are by default **published**, provided the class is compiled in the `{$M+}` state or is derived from a class compiled in the `{$M+}` state; otherwise, such members are **public**.

For readability, it is best to organize a class declaration by visibility, placing all the **private** members together, followed by all the **protected** members, and so forth. This way each visibility reserved word appears at most once and marks the beginning of a new 'section' of the declaration. So a typical class declaration should like this:

```
type
    TMyClass = class(TControl)
      private
         ...  { private declarations here }
      protected
         ...  { protected declarations here }
      public
         ...  { public declarations here }
      published
         ...  { published declarations here }
    end;
```

You can increase the visibility of a member in a descendant class by redeclaring it, but you cannot decrease its visibility. For example, a **protected** property can be made **public** in a descendant, but not **private**. Moreover, **published** members cannot become **public** in a descendant class. For more information, see Property overrides and redeclarations (see page 530).

Private, Protected, and Public Members

A **private** member is invisible outside of the unit or program where its class is declared. In other words, a **private** method cannot be called from another module, and a **private** field or property cannot be read or written to from another module. By placing related class declarations in the same module, you can give the classes access to one another's **private** members without making those members more widely accessible.

A **protected** member is visible anywhere in the module where its class is declared and from any descendant class, regardless of the module where the descendant class appears. A **protected** method can be called, and a **protected** field or property read or written to, from the definition of any method belonging to a class that descends from the one where the **protected** member is declared. Members that are intended for use only in the implementation of derived classes are usually protected.

A **public** member is visible wherever its class can be referenced.

Strict Visibility Specifiers

In addition to **private** and **protected** visibility specifiers, the Delphi compiler supports additional visibility settings with greater access constraints. These settings are **strict private** and **strict protected** visibility. These settings strictly comply with the .NET Common Language Specification (CLS), and they can also be used in Win32 applications.

Class members with **strict private** visibility are accessible only within the class in which they are declared. They are not visible to procedures or functions declared within the same unit. Class members with **strict protected** visibility are visible within the class in which they are declared, and within any descendant class, regardless of where it is declared. Furthermore, when instance members (those declared without the **class** or **class var** keywords) are declared **strict private** or **strict protected**, they are inaccessible outside of the instance of a class in which they appear. An instance of a class cannot access **strict protected** or **strict protected** instance members in other instances of the same class.

Delphi's traditional **private** visibility specifier maps to the CLR's `assembly` visibility. Delphi's **protected** visibility specifier maps to the CLR's `assembly` or `family` visibility.

Note: The word *strict* is treated as a directive within the context of a class declaration. Within a class declaration you cannot declare a member named 'strict', but it is acceptable for use outside of a class declaration.

Published Members

Published members have the same visibility as public members. The difference is that runtime type information (RTTI) is generated for published members. RTTI allows an application to query the fields and properties of an object dynamically and to locate its methods. RTTI is used to access the values of properties when saving and loading form files, to display properties in the **Object Inspector**, and to associate specific methods (called event handlers) with specific properties (called events).

Published properties are restricted to certain data types. Ordinal, string, class, interface, variant, and method-pointer types can be published. So can set types, provided the upper and lower bounds of the base type have ordinal values between 0 and 31. (In other words, the set must fit in a byte, word, or double word.) Any real type except **Real48** can be published. Properties of an array type (as distinct from array properties, discussed below) cannot be published.

Some properties, although publishable, are not fully supported by the streaming system. These include properties of record types, array properties (see page 530) of all publishable types, and properties of enumerated types (see page 554) that include anonymous values. If you publish a property of this kind, the **Object Inspector** won't display it correctly, nor will the property's value be preserved when objects are streamed to disk.

All methods are publishable, but a class cannot publish two or more overloaded methods with the same name. Fields can be published only if they are of a class or interface type.

A class cannot have published members unless it is compiled in the `{$M+}` state or descends from a class compiled in the `{$M+}` state. Most classes with published members derive from TPersistent, which is compiled in the `{$M+}` state, so it is seldom necessary to use the `$M` directive.

Note: Identifiers that contain Unicode characters are not allowed in published sections of classes, or in types used by published members.

Automated Members (Win32 Only)

Automated members have the same visibility as public members. The difference is that Automation type information (required for Automation servers) is generated for automated members. Automated members typically appear only in Win32 classes, and the automated reserved word has been deprecated in the .NET compiler. The **automated** reserved word is maintained for backward compatibility. The TAutoObject class in the ComObj unit does not use **automated**.

The following restrictions apply to methods and properties declared as automated.

- The types of all properties, array property parameters, method parameters, and function results must be automatable. The automatable types are **Byte**, **Currency**, **Real**, **Double**, **Longint**, **Integer**, **Single**, **Smallint**, **AnsiString**, **WideString**, **TDateTime**, **Variant**, **OleVariant**, **WordBool**, and all interface types.
- Method declarations must use the default **register** calling convention. They can be virtual, but not dynamic.
- Property declarations can include access specifiers (**read** and **write**) but other specifiers (**index**, **stored**, **default**, and **nodefault**) are not allowed. Access specifiers must list a method identifier that uses the default **register** calling convention; field identifiers are not allowed.
- Property declarations must specify a type. Property overrides are not allowed.

The declaration of an automated method or property can include a **dispid** directive. Specifying an already used ID in a **dispid** directive causes an error.

On the Win32 platform, this directive must be followed by an integer constant that specifies an Automation dispatch ID for the member. Otherwise, the compiler automatically assigns the member a dispatch ID that is one larger than the largest dispatch ID used by any method or property in the class and its ancestors. For more information about Automation (on Win32 only), see Automation objects (see page 633).

Forward Declarations and Mutually Dependent Classes

If the declaration of a class type ends with the word **class** and a semicolon - that is, if it has the form

type *className* = **class**;

with no ancestor or class members listed after the word **class**, then it is a forward declaration. A forward declaration must be resolved by a defining declaration of the same class within the same type declaration section. In other words, between a forward declaration and its defining declaration, nothing can occur except other type declarations.

Forward declarations allow mutually dependent classes. For example,

```
type
      TFigure = class;   // forward declaration
      TDrawing = class
         Figure: TFigure;
           ...
      end;

      TFigure = class   // defining declaration
         Drawing: TDrawing;
           ...
      end;
```

Do not confuse forward declarations with complete declarations of types that derive from TObject without declaring any class members.

```
type
     TFirstClass = class;   // this is a forward declaration
     TSecondClass = class   // this is a complete class declaration
     end;                   //
     TThirdClass = class(TObject);  // this is a complete class declaration
```

See Also

Fields (see page 519)

Methods (see page 521)

Properties (see page 530)

Events (see page 536)

Class References (see page 539)

Exceptions (see page 541)

Nested Type Declarations (see page 546)

Operator Overloading (see page 548)

Class Helpers (see page 550)

3.1.3.1.2 Fields

This topic describes the syntax of class data fields declarations.

About Fields

A field is like a variable that belongs to an object. Fields can be of any type, including class types. (That is, fields can hold object references.) Fields are usually private.

To define a field member of a class, simply declare the field as you would a variable. For example, the following declaration creates a class called `TNumber` whose only member, other than the methods is inherits from TObject, is an integer field called `Int`.

```
type
  TNumber = class
    var
      Int: Integer;
  end;
```

The **var** keyword is optional. However, if it is not used, then all field declarations must occur before any property or method declarations. After any property or method declarations, the **var** may be used to introduce any additional field declarations.

Fields are statically bound; that is, references to them are fixed at compile time. To see what this means, consider the following code.

```
type
   TAncestor = class
      Value: Integer;
   end;

   TDescendant = class(TAncestor)
      Value: string;    // hides the inherited Value field
   end;

var
   MyObject: TAncestor;

begin
   MyObject := TDescendant.Create;
   MyObject.Value := 'Hello!'     // error

  (MyObject as TDescendant).Value := 'Hello!'   // works!
end;
```

Although `MyObject` holds an instance of `TDescendant`, it is declared as `TAncestor`. The compiler therefore interprets `MyObject.Value` as referring to the (integer) field declared in `TAncestor`. Both fields, however, exist in the `TDescendant` object; the inherited `Value` is hidden by the new one, and can be accessed through a typecast.

Constants (see page 589), and typed constant (see page 589) declarations can appear in classes and non-anonymous records at global scope. Both constants and typed constants can also appear within nested type (see page 546) definitions. Constants and typed constants can appear only within class definitions when the class is defined locally to a procedure (i.e. they cannot appear within records defined within a procedure).

Class Fields

Class fields are data fields in a class that can be accessed without an object reference (unlike the normal “instance fields” which are discussed above). The data stored in a class field are shared by all instances of the class and may be accessed by referring to the class or to a variable that represents an instance of the class.

You can introduce a block of class fields within a class declaration by using the **class var** block declaration. All fields declared after **class var** have static storage attributes. A **class var** block is terminated by the following:

1. Another **class var** or **var** declaration
2. A procedure or function (i.e. method) declaration (see page 521) (including class procedures and class functions)
3. A property declaration (see page 530) (including class properties)
4. A constructor or destructor declaration (see page 521)
5. A visibility scope specifier (see page 514) (**public**, **private**, **protected**, **published**, **strict private**, and **strict protected**)

For example:

```
type
    TMyClass = class
      public
        class var        // Introduce a block of class static fields.
          Red: Integer;
          Green: Integer;
          Blue: Integer;
        var              // Ends the class var block.
          InstanceField: Integer;
    end;
```

The class fields `Red`, `Green`, and `Blue` can be accessed with the code:

```
TMyClass.Red := 1;
TMyClass.Green := 2;
TMyClass.Blue := 3;
```

Class fields may also be accessed through an instance of the class. With the following declaration:

```
var
    myObject: TMyClass;
```

This code has the same effect as the assignments to `Red`, `Green`, and `Blue` above:

```
myObject.Red := 1;
myObject.Green := 2;
myObject.Blue := 3;
```

See Also

Classes and Objects (↗ see page 514)

Methods (↗ see page 521)

Properties (↗ see page 530)

Nested Type Declarations (↗ see page 546)

Class References (↗ see page 539)

Exceptions (↗ see page 541)

Operator Overloading (↗ see page 548)

Class Helpers (↗ see page 550)

3.1.3.1.3 Methods

A method is a procedure or function associated with a class. A call to a method specifies the object (or, if it is a class method, the class) that the method should operate on. For example, `SomeObject.Free` calls the `Free` method in `SomeObject`.

This topic covers the following material:

- Methods declarations and implementation
- Method binding
- Overloading methods
- Constructors and destructors
- Message methods

About Methods

Within a class declaration, methods appear as procedure and function headings, which work like **forward** declarations. Somewhere after the class declaration, but within the same module, each method must be implemented by a defining declaration. For example, suppose the declaration of `TMyClass` includes a method called `DoSomething`:

```
type
   TMyClass = class(TObject)
      ...
      procedure DoSomething;
      ...
   end;
```

A defining declaration for `DoSomething` must occur later in the module:

```
procedure TMyClass.DoSomething;
```

```
begin
     ...
end;
```

While a class can be declared in either the interface or the implementation section of a unit, defining declarations for a class' methods must be in the implementation section.

In the heading of a defining declaration, the method name is always qualified with the name of the class to which it belongs. The heading can repeat the parameter list from the class declaration; if it does, the order, type and names of the parameters must match exactly, and if the method is a function, the return value must match as well.

Method declarations can include special directives that are not used with other functions or procedures. Directives should appear in the class declaration only, not in the defining declaration, and should always be listed in the following order:

reintroduce; **overload**; *binding;calling convention;***abstract**; *warning*

where *binding* is **virtual**, **dynamic**, or **override**; calling convention is **register**, **pascal**, **cdecl**, **stdcall**, or **safecall**; and *warning* is **platform**, **deprecated**, or **library**.

Inherited

The reserved word **inherited** plays a special role in implementing polymorphic behavior. It can occur in method definitions, with or without an identifier after it.

If **inherited** is followed by the name of a member, it represents a normal method call or reference to a property or field - except that the search for the referenced member begins with the immediate ancestor of the enclosing method's class. For example, when

```
inherited Create(...);
```

occurs in the definition of a method, it calls the inherited `Create`.

When **inherited** has no identifier after it, it refers to the inherited method with the same name as the enclosing method or, if the enclosing method is a message handler, to the inherited message handler for the same message. In this case, **inherited** takes no explicit parameters, but passes to the inherited method the same parameters with which the enclosing method was called. For example,

```
inherited;
```

occurs frequently in the implementation of constructors. It calls the inherited constructor with the same parameters that were passed to the descendant.

Self

Within the implementation of a method, the identifier **Self** references the object in which the method is called. For example, here is the implementation of TCollection's `Add` method in the `Classes` unit.

```
function TCollection.Add: TCollectionItem;
begin
    Result := FItemClass.Create(Self);
end;
```

3

The `Add` method calls the `Create` method in the class referenced by the `FItemClass` field, which is always a TCollectionItem descendant. `TCollectionItem.Create` takes a single parameter of type TCollection, so `Add` passes it the TCollection instance object where `Add` is called. This is illustrated in the following code.

```
var MyCollection: TCollection;
    ...
    MyCollection.Add   // MyCollection is passed to the TCollectionItem.Create method
```

Self is useful for a variety of reasons. For example, a member identifier declared in a class type might be redeclared in the block of one of the class' methods. In this case, you can access the original member identifier as `Self.Identifier`.

For information about **Self** in class methods, see Class methods (see page 539).

Method Binding

Method bindings can be static (the default), **virtual**, or **dynamic**. Virtual and dynamic methods can be overridden, and they can be abstract. These designations come into play when a variable of one class type holds a value of a descendant class type. They determine which implementation is activated when a method is called.

Static Methods

Methods are by default static. When a static method is called, the declared (compile-time) type of the class or object variable used in the method call determines which implementation to activate. In the following example, the `Draw` methods are static.

```
type
    TFigure = class
      procedure Draw;
    end;

    TRectangle = class(TFigure)
      procedure Draw;
    end;
```

Given these declarations, the following code illustrates the effect of calling a static method. In the second call to `Figure.Draw`, the `Figure` variable references an object of class `TRectangle`, but the call invokes the implementation of `Draw` in `TFigure`, because the declared type of the `Figure` variable is `TFigure`.

```
var
    Figure: TFigure;
    Rectangle: TRectangle;

    begin
            Figure := TFigure.Create;
            Figure.Draw;              // calls TFigure.Draw
            Figure.Destroy;
            Figure := TRectangle.Create;
            Figure.Draw;              // calls TFigure.Draw

            TRectangle(Figure).Draw;  // calls TRectangle.Draw

            Figure.Destroy;
            Rectangle := TRectangle.Create;
            Rectangle.Draw;          // calls TRectangle.Draw
            Rectangle.Destroy;
    end;
```

Virtual and Dynamic Methods

To make a method **virtual** or **dynamic**, include the **virtual** or **dynamic** directive in its declaration. Virtual and dynamic methods, unlike static methods, can be overridden in descendant classes. When an overridden method is called, the actual (runtime) type of the class or object used in the method call—not the declared type of the variable—determines which implementation to activate.

To override a method, redeclare it with the **override** directive. An **override** declaration must match the ancestor declaration in the order and type of its parameters and in its result type (if any).

In the following example, the `Draw` method declared in `TFigure` is overridden in two descendant classes.

```
type
    TFigure = class
      procedure Draw; virtual;
    end;
    TRectangle = class(TFigure)
      procedure Draw; override;
    end;

    TEllipse = class(TFigure)
      procedure Draw; override;
    end;
```

Given these declarations, the following code illustrates the effect of calling a virtual method through a variable whose actual type varies at runtime.

```
var
   Figure: TFigure;

   begin
     Figure := TRectangle.Create;
     Figure.Draw;      // calls TRectangle.Draw
     Figure.Destroy;
     Figure := TEllipse.Create;
     Figure.Draw;      // calls TEllipse.Draw
     Figure.Destroy;
   end;
```

Only **virtual** and **dynamic** methods can be overridden. All methods, however, can be overloaded; see Overloading methods.

The Delphi compiler also supports the concept of a *final* virtual method. When the keyword **final** is applied to a virtual method, no descendent class can override that method. Use of the final keyword is an important design decision that can help document how the class is intended to be used. It can also give the compiler hints that allow it to optimize the code it produces.

Virtual Versus Dynamic

In Delphi for .NET, **virtual** and **dynamic** methods are identical. In Delphi for Win32, virtual and dynamic methods are semantically equivalent. However, they differ in the implementation of method-call dispatching at runtime: virtual methods optimize for speed, while dynamic methods optimize for code size.

In general, virtual methods are the most efficient way to implement polymorphic behavior. Dynamic methods are useful when a base class declares many overridable methods which are inherited by many descendant classes in an application, but only occasionally overridden.

Note: Only use dynamic methods if there is a clear, observable benefit. Generally, use virtual methods.

Overriding Versus Hiding

If a method declaration specifies the same method identifier and parameter signature as an inherited method, but doesn't include **override**, the new declaration merely hides the inherited one without overriding it. Both methods exist in the descendant class, where the method name is statically bound. For example,

```
type
   T1 = class(TObject)
      procedure Act; virtual;
   end;

   T2 = class(T1)
      procedure Act;   // Act is redeclared, but not overridden
   end;

var
   SomeObject: T1;

begin
   SomeObject := T2.Create;
   SomeObject.Act;    // calls T1.Act
end;
```

Reintroduce

The **reintroduce** directive suppresses compiler warnings about hiding previously declared virtual methods. For example,

```
procedure DoSomething; reintroduce;   // the ancestor class also has a DoSomething method
```

Use **reintroduce** when you want to hide an inherited virtual method with a new one.

Abstract Methods

An abstract method is a virtual or dynamic method that has no implementation in the class where it is declared. Its implementation is deferred to a descendant class. Abstract methods must be declared with the directive **abstract** after **virtual** or **dynamic**. For example,

```
procedure DoSomething; virtual; abstract;
```

You can call an **abstract** method only in a class or instance of a class in which the method has been overridden.

Note: The Delphi for .NET compiler allows an entire class to be declared abstract, even though it does not contain any virtual abstract methods. See Class Types (↗ see page 514) for more information.

Class Methods

Most methods are called instance methods, because they operate on an individual instance of an object. A class method is a method (other than a constructor) that operates on classes instead of objects. There are two types of class methods: ordinary class methods and class static methods.

Ordinary Class Methods

The definition of a class method must begin with the reserved word **class**. For example,

```
type
  TFigure = class
  public
     class function Supports(Operation: string): Boolean; virtual;
     class procedure GetInfo(var Info: TFigureInfo); virtual;
     ...
  end;
```

The defining declaration of a class method must also begin with **class**. For example,

```
class procedure TFigure.GetInfo(var Info: TFigureInfo);
begin
    ...
end;
```

In the defining declaration of a class method, the identifier **Self** represents the class where the method is called (which could be a descendant of the class in which it is defined). If the method is called in the class `C`, then **Self** is of the type class of `C`. Thus you cannot use the **Self** to access instance fields, instance properties, and normal (object) methods, but you can use it to call constructors and other class methods, or to access class properties and class fields.

A class method can be called through a class reference or an object reference. When it is called through an object reference, the class of the object becomes the value of **Self**.

Class Static Methods

Like class methods, class static methods can be accessed without an object reference. Unlike ordinary class methods, class static methods have no **Self** parameter at all. They also cannot access any instance members. (They still have access to class fields, class properties, and class methods.) Also unlike class methods, class static methods cannot be declared **virtual**.

Methods are made class static by appending the word **static** to their declaration, for example

```
type
   TMyClass = class
     strict private
       class var
       FX: Integer;

     strict protected

       // Note: accessors for class properties must be declared class static.
       class function GetX: Integer; static;
       class procedure SetX(val: Integer); static;

     public
       class property X: Integer read GetX write SetX;
       class procedure StatProc(s: String); static;
   end;
```

Like a class method, you can call a class static method through the class type (i.e. without having an object reference), for example

```
TMyClass.X := 17;
TMyClass.StatProc('Hello');
```

Overloading Methods

A method can be redeclared using the **overload** directive. In this case, if the redeclared method has a different parameter signature from its ancestor, it overloads the inherited method without hiding it. Calling the method in a descendant class activates whichever implementation matches the parameters in the call.

If you overload a virtual method, use the **reintroduce** directive when you redeclare it in descendant classes. For example,

```
type
  T1 = class(TObject)
    procedure Test(I: Integer); overload; virtual;
  end;

  T2 = class(T1)
    procedure Test(S: string); reintroduce; overload;
  end;
  ...

SomeObject := T2.Create;
SomeObject.Test('Hello!');       // calls T2.Test
SomeObject.Test(7);              // calls T1.Test
```

Within a class, you cannot publish multiple overloaded methods with the same name. Maintenance of runtime type information requires a unique name for each published member.

```
type
    TSomeClass = class
      published
        function Func(P: Integer): Integer;
        function Func(P: Boolean): Integer;   // error
          ...
```

Methods that serve as property **read** or **write** specifiers cannot be overloaded.

The implementation of an overloaded method must repeat the parameter list from the class declaration. For more information about overloading, see Overloading procedures and functions (see page 662).

Constructors

A constructor is a special method that creates and initializes instance objects. The declaration of a constructor looks like a procedure declaration, but it begins with the word **constructor**. Examples:

```
constructor Create;
constructor Create(AOwner: TComponent);
```

Constructors must use the default **register** calling convention. Although the declaration specifies no return value, a constructor returns a reference to the object it creates or is called in.

A class can have more than one constructor, but most have only one. It is conventional to call the constructor `Create`.

To create an object, call the constructor method on a class type. For example,

```
MyObject := TMyClass.Create;
```

This allocates storage for the new object, sets the values of all ordinal fields to zero, assigns **nil** to all pointer and class-type fields, and makes all string fields empty. Other actions specified in the constructor implementation are performed next; typically, objects are initialized based on values passed as parameters to the constructor. Finally, the constructor returns a reference to the newly allocated and initialized object. The type of the returned value is the same as the class type specified in the constructor call.

If an exception is raised during execution of a constructor that was invoked on a class reference, the `Destroy` destructor is automatically called to destroy the unfinished object.

When a constructor is called using an object reference (rather than a class reference), it does not create an object. Instead, the constructor operates on the specified object, executing only the statements in the constructor's implementation, and then returns a reference to the object. A constructor is typically invoked on an object reference in conjunction with the reserved word **inherited** to execute an inherited constructor.

Here is an example of a class type and its constructor.

```
type
   TShape = class(TGraphicControl)
     private
       FPen: TPen;
       FBrush: TBrush;
       procedure PenChanged(Sender: TObject);
       procedure BrushChanged(Sender: TObject);
     public
       constructor Create(Owner: TComponent); override;
       destructor Destroy; override;
       ...
   end;

constructor TShape.Create(Owner: TComponent);
begin
    inherited Create(Owner);     // Initialize inherited parts
    Width := 65;          // Change inherited properties
    Height := 65;
    FPen := TPen.Create;  // Initialize new fields
    FPen.OnChange := PenChanged;
    FBrush := TBrush.Create;
    FBrush.OnChange := BrushChanged;
end;
```

The first action of a constructor is usually to call an inherited constructor to initialize the object's inherited fields. The constructor then initializes the fields introduced in the descendant class. Because a constructor always clears the storage it allocates for a new object, all fields start with a value of zero (ordinal types), **nil** (pointer and class types), empty (string types), or Unassigned (variants). Hence there is no need to initialize fields in a constructor's implementation except to nonzero or nonempty values.

When invoked through a class-type identifier, a constructor declared as **virtual** is equivalent to a static constructor. When combined with class-reference types, however, virtual constructors allow polymorphic construction of objectsthat is, construction of objects whose types aren't known at compile time. (See Class references (↗ see page 539).)

Note: For more information on constructors, destructors, and memory management issues on the .NET platform, please see the topic Memory Management Issues on the .NET Platform (↗ see page 653).

The Class Constructor (.NET)

A class constructor executes before a class is referenced or used. The class constructor may be declared as **public** or **private**. There can be at most one class constructor declared in a class. Descendants can declare their own class constructor, however, do not call **inherited** within the body of a class constructor. In fact, you cannot call a class constructor directly, or access it in any way (such as taking its address). The compiler generates code to call class constructors for you.

There can be no guarantees on when a class constructor will execute, except to say that it will execute at some time before the class is used. On the .NET platform in order for a class to be "used", it must reside in code that is actually executed. For example, if a class is first referenced in an **if** statement, and the test of the **if** statement is never true during the course of execution, then the class will never be loaded and JIT compiled. Hence, in this case the class constructor would not be called.

The following class declaration demonstrates the syntax of class properties and fields, as well as class static methods and class constructors:

```
type
    TMyClass = class
      strict protected

        // Accessors for class properties must be declared class static.
        class function GetX: Integer; static;
        class procedure SetX(val: Integer); static;
      public
        class property X: Integer read GetX write SetX;
        class procedure StatProc(s: String); static;
      strict private
        class var
          FX: Integer;
          class constructor Create;
    end;
```

Destructors

A destructor is a special method that destroys the object where it is called and deallocates its memory. The declaration of a destructor looks like a procedure declaration, but it begins with the word **destructor**. Example:

```
destructor SpecialDestructor(SaveData: Boolean);
destructor Destroy; override;
```

Destructors on Win32 must use the default **register** calling convention. Although a class can have more than one destructor, it is recommended that each class override the inherited `Destroy` method and declare no other destructors.

To call a destructor, you must reference an instance object. For example,

```
MyObject.Destroy;
```

When a destructor is called, actions specified in the destructor implementation are performed first. Typically, these consist of destroying any embedded objects and freeing resources that were allocated by the object. Then the storage that was allocated for the object is disposed of.

Here is an example of a destructor implementation.

```
destructor TShape.Destroy;
begin
    FBrush.Free;
    FPen.Free;
    inherited Destroy;
end;
```

The last action in a destructor's implementation is typically to call the inherited destructor to destroy the object's inherited fields.

When an exception is raised during creation of an object, Destroy is automatically called to dispose of the unfinished object. This means that `Destroy` must be prepared to dispose of partially constructed objects. Because a constructor sets the fields of a new object to zero or empty values before performing other actions, class-type and pointer-type fields in a partially constructed object are always **nil**. A destructor should therefore check for **nil** values before operating on class-type or pointer-type fields. Calling the Free method (defined in TObject), rather than `Destroy`, offers a convenient way of checking for **nil** values before destroying an object.

Note: For more information on constructors, destructors, and memory management issues on the .NET platform, please see the topic Memory Management Issues on the .NET Platform (see page 653).

Message Methods

Message methods implement responses to dynamically dispatched messages. The message method syntax is supported on all platforms. VCL uses message methods to respond to Windows messages.

A message method is created by including the **message** directive in a method declaration, followed by an integer constant between 1 and 49151 which specifies the message ID. For message methods in VCL controls, the integer constant can be one of the Win32 message IDs defined, along with corresponding record types, in the `Messages` unit. A message method must be a procedure that takes a single **var** parameter.

For example:

```
type
    TTextBox = class(TCustomControl)
      private
       procedure WMChar(var Message: TWMChar); message WM_CHAR;
       ...
    end;
```

A message method does not have to include the **override** directive to override an inherited message method. In fact, it doesn't have to specify the same method name or parameter type as the method it overrides. The message ID alone determines which message the method responds to and whether it is an override.

Implementing Message Methods

The implementation of a message method can call the inherited message method, as in this example:

```
procedure TTextBox.WMChar(var Message: TWMChar);
begin
   if Message.CharCode = Ord(#13) then
      ProcessEnter
   else
      inherited;
end;
```

The **inherited** statement searches backward through the class hierarchy and invokes the first message method with the same ID as the current method, automatically passing the message record to it. If no ancestor class implements a message method for the given ID, **inherited** calls the `DefaultHandler` method originally defined in TObject.

The implementation of `DefaultHandler` in TObject simply returns without performing any actions. By overriding `DefaultHandler`, a class can implement its own default handling of messages. On Win32, the `DefaultHandler` method for controls calls the Win32 API `DefWindowProc`.

Message Dispatching

Message handlers are seldom called directly. Instead, messages are dispatched to an object using the `Dispatch` method inherited from TObject:

```
procedure Dispatch(var Message);
```

The `Message` parameter passed to `Dispatch` must be a record whose first entry is a field of type **Word** containing a message ID.

`Dispatch` searches backward through the class hierarchy (starting from the class of the object where it is called) and invokes the first message method for the ID passed to it. If no message method is found for the given ID, `Dispatch` calls

`DefaultHandler`.

See Also

Classes and Objects (↗ see page 514)

Fields (↗ see page 519)

Properties (↗ see page 530)

Nested Type Declarations (↗ see page 546)

Class References (↗ see page 539)

Exceptions (↗ see page 541)

Operator Overloading (↗ see page 548)

Class Helpers (↗ see page 550)

3.1.3.1.4 Properties

This topic describes the following material:

- Property access
- Array properties
- Index specifiers
- Storage specifiers
- Property overrides and redeclarations
- Class properties

About Properties

A property, like a field, defines an attribute of an object. But while a field is merely a storage location whose contents can be examined and changed, a property associates specific actions with reading or modifying its data. Properties provide control over access to an object's attributes, and they allow attributes to be computed.

The declaration of a property specifies a name and a type, and includes at least one access specifier. The syntax of a property declaration is

property *propertyName[indexes]: type* `index` *integerConstant specifiers;*

where

- *propertyName* is any valid identifier.
- *[indexes]* is optional and is a sequence of parameter declarations separated by semicolons. Each parameter declaration has the form *identifier1*, ..., *identifiern*: type. For more information, see Array Properties, below.
- type must be a predefined or previously declared type identifier. That is, property declarations like `property Num: 0..9` ... are invalid.
- the index *integerConstant* clause is optional. For more information, see Index Specifiers, below.
- specifiers is a sequence of **read**, **write**, **stored**, **default** (or **nodefault**), and **implements** specifiers. Every property declaration must have at least one **read** or **write** specifier.

Properties are defined by their access specifiers. Unlike fields, properties cannot be passed as **var** parameters, nor can the @ operator be applied to a property. The reason is that a property doesn't necessarily exist in memory. It could, for instance, have a **read** method that retrieves a value from a database or generates a random value.

Property Access

Every property has a **read** specifier, a **write** specifier, or both. These are called access specifiers and they have the form

read *fieldOrMethod*

write *fieldOrMethod*

where *fieldOrMethod* is the name of a field or method declared in the same class as the property or in an ancestor class.

- If *fieldOrMethod* is declared in the same class, it must occur before the property declaration. If it is declared in an ancestor class, it must be visible from the descendant; that is, it cannot be a private field or method of an ancestor class declared in a different unit.
- If *fieldOrMethod* is a field, it must be of the same type as the property.
- If *fieldOrMethod* is a method, it cannot be **dynamic** and, if **virtual**, cannot be overloaded. Moreover, access methods for a published property must use the default **register** calling convention.
- In a **read** specifier, if **fieldOrMethod** is a method, it must be a parameterless function whose result type is the same as the property's type. (An exception is the access method for an indexed property or an array property.)
- In a **write** specifier, if *fieldOrMethod* is a method, it must be a procedure that takes a single value or **const** parameter of the same type as the property (or more, if it is an array property or indexed property).

For example, given the declaration

```
property Color: TColor read GetColor write SetColor;
```

the `GetColor` method must be declared as

```
function GetColor: TColor;
```

and the `SetColor` method must be declared as one of these:

```
procedure SetColor(Value: TColor);
procedure SetColor(const Value: TColor);
```

(The name of `SetColor`'s parameter, of course, doesn't have to be `Value`.)

When a property is referenced in an expression, its value is read using the field or method listed in the **read** specifier. When a property is referenced in an assignment statement, its value is written using the field or method listed in the **write** specifier.

The example below declares a class called `TCompass` with a published property called `Heading`. The value of `Heading` is read through the `FHeading` field and written through the `SetHeading` procedure.

```
type
   THeading = 0..359;
   TCompass = class(TControl)
     private
        FHeading: THeading;
        procedure SetHeading(Value: THeading);
     published
        property Heading: THeading read FHeading write SetHeading;
        ...
    end;
```

Given this declaration, the statements

```
if Compass.Heading = 180 then GoingSouth;
Compass.Heading := 135;
```

correspond to

```
if Compass.FHeading = 180 then GoingSouth;
Compass.SetHeading(135);
```

In the `TCompass` class, no action is associated with reading the `Heading` property; the **read** operation consists of retrieving the value stored in the `FHeading` field. On the other hand, assigning a value to the `Heading` property translates into a call to the

`SetHeading` method, which, presumably, stores the new value in the `FHeading` field as well as performing other actions. For example, `SetHeading` might be implemented like this:

```
procedure TCompass.SetHeading(Value: THeading);
begin
            if FHeading <> Value then
             begin
               FHeading := Value;
               Repaint;    // update user interface to reflect new value
             end;
end;
```

A property whose declaration includes only a **read** specifier is a read-only property, and one whose declaration includes only a **write** specifier is a write-only property. It is an error to assign a value to a read-only property or use a write-only property in an expression.

Array Properties

Array properties are indexed properties. They can represent things like items in a list, child controls of a control, and pixels of a bitmap.

The declaration of an array property includes a parameter list that specifies the names and types of the indexes. For example,

```
property Objects[Index: Integer]: TObject read GetObject write SetObject;
property Pixels[X, Y: Integer]: TColor read GetPixel write SetPixel;
property Values[const Name: string]: string read GetValue write SetValue;
```

The format of an index parameter list is the same as that of a procedure's or function's parameter list, except that the parameter declarations are enclosed in brackets instead of parentheses. Unlike arrays, which can use only ordinal-type indexes, array properties allow indexes of any type.

For array properties, access specifiers must list methods rather than fields. The method in a **read** specifier must be a function that takes the number and type of parameters listed in the property's index parameter list, in the same order, and whose result type is identical to the property's type. The method in a **write** specifier must be a procedure that takes the number and type of parameters listed in the property's index parameter list, in the same order, plus an additional value or **const** parameter of the same type as the property.

For example, the access methods for the array properties above might be declared as

```
function GetObject(Index: Integer): TObject;
function GetPixel(X, Y: Integer): TColor;
function GetValue(const Name: string): string;
procedure SetObject(Index: Integer; Value: TObject);
procedure SetPixel(X, Y: Integer; Value: TColor);
procedure SetValue(const Name, Value: string);
```

An array property is accessed by indexing the property identifier. For example, the statements

```
if Collection.Objects[0] = nil then Exit;
Canvas.Pixels[10, 20] := clRed;
Params.Values['PATH'] := 'C:\BIN';
```

correspond to

```
if Collection.GetObject(0) = nil then Exit;
Canvas.SetPixel(10, 20, clRed);
Params.SetValue('PATH', 'C:\BIN');
```

The definition of an array property can be followed by the **default** directive, in which case the array property becomes the default property of the class. For example,

```
type
   TStringArray = class
    public
       property Strings[Index: Integer]: string ...; default;
          ...
    end;
```

If a class has a default property, you can access that property with the abbreviation `object[index]`, which is equivalent to `object.property[index]`. For example, given the declaration above, `StringArray.Strings[7]` can be abbreviated to `StringArray[7]`. A class can have only one default property with a given signature (array parameter list), but it is possible to overload the default property. Changing or hiding the default property in descendant classes may lead to unexpected behavior, since the compiler always binds to properties statically.

Index Specifiers

Index specifiers allow several properties to share the same access method while representing different values. An index specifier consists of the directive **index** followed by an integer constant between -2147483647 and 2147483647. If a property has an index specifier, its **read** and **write** specifiers must list methods rather than fields. For example,

```
type
   TRectangle = class
     private
       FCoordinates: array[0..3] of Longint;
       function GetCoordinate(Index: Integer): Longint;
       procedure SetCoordinate(Index: Integer; Value: Longint);
     public
       property Left: Longint index 0 read GetCoordinate write SetCoordinate;
       property Top: Longint index 1 read GetCoordinate write SetCoordinate;
       property Right: Longint index 2 read GetCoordinate write SetCoordinate;
       property Bottom: Longint index 3 read GetCoordinate write SetCoordinate;
       property Coordinates[Index: Integer]: Longint read GetCoordinate write SetCoordinate;
       ...
   end;
```

An access method for a property with an index specifier must take an extra value parameter of type Integer. For a **read** function, it must be the last parameter; for a **write** procedure, it must be the second-to-last parameter (preceding the parameter that specifies the property value). When a program accesses the property, the property's integer constant is automatically passed to the access method.

Given the declaration above, if `Rectangle` is of type `TRectangle`, then

```
Rectangle.Right := Rectangle.Left + 100;
```

corresponds to

```
Rectangle.SetCoordinate(2, Rectangle.GetCoordinate(0) + 100);
```

Storage Specifiers

The optional **stored**, **default**, and **nodefault** directives are called storage specifiers. They have no effect on program behavior, but control whether or not to save the values of published properties in form files.

The **stored** directive must be followed by **True**, **False**, the name of a **Boolean** field, or the name of a parameterless method that returns a **Boolean** value. For example,

```
property Name: TComponentName read FName write SetName stored False;
```

If a property has no **stored** directive, it is treated as if stored **True** were specified.

The **default** directive must be followed by a constant of the same type as the property. For example,

```
property Tag: Longint read FTag write FTag default 0;
```

To override an inherited **default** value without specifying a new one, use the **nodefault** directive. The **default** and **nodefault** directives are supported only for ordinal types and for set types, provided the upper and lower bounds of the set's base type have ordinal values between 0 and 31; if such a property is declared without **default** or **nodefault**, it is treated as if **nodefault** were specified. For reals, pointers, and strings, there is an implicit **default** value of 0, **nil**, and `''` (the empty string), respectively.

Note: You can't use the ordinal value 2147483648 has a default value. This value is used internally to represent nodefault

. When saving a component's state, the storage specifiers of the component's published properties are checked. If a property's current value is different from its **default** value (or if there is no **default** value) and the **stored** specifier is **True**, then the property's value is saved. Otherwise, the property's value is not saved.

Note: Property values are not automatically initialized to the default value. That is, the default directive controls only when property values are saved to the form file, but not the initial value of the property on a newly created instance.

Storage specifiers are not supported for array properties. The **default** directive has a different meaning when used in an array property declaration. See Array Properties, above.

Property Overrides and Redeclarations

A property declaration that doesn't specify a type is called a property override. Property overrides allow you to change a property's inherited visibility or specifiers. The simplest override consists only of the reserved word **property** followed by an inherited property identifier; this form is used to change a property's visibility. For example, if an ancestor class declares a property as protected, a derived class can redeclare it in a public or published section of the class. Property overrides can include **read**, **write**,**stored**, **default**, and **nodefault** directives; any such directive overrides the corresponding inherited directive. An override can replace an inherited access specifier, add a missing specifier, or increase a property's visibility, but it cannot remove an access specifier or decrease a property's visibility. An override can include an **implements** directive, which adds to the list of implemented interfaces without removing inherited ones.

The following declarations illustrate the use of property overrides.

```
type
   TAncestor = class
       ...
     protected
       property Size: Integer read FSize;
       property Text: string read GetText write SetText;
       property Color: TColor read FColor write SetColor stored False;
       ...
   end;

type

   TDerived = class(TAncestor)
       ...
     protected
       property Size write SetSize;
     published
       property Text;
       property Color stored True default clBlue;
       ...
   end;
```

The override of `Size` adds a **write** specifier to allow the property to be modified. The overrides of `Text` and `Color` change the visibility of the properties from protected to published. The property override of `Color` also specifies that the property should be filed if its value isn't `clBlue`.

A redeclaration of a property that includes a type identifier hides the inherited property rather than overriding it. This means that a new property is created with the same name as the inherited one. Any property declaration that specifies a type must be a complete declaration, and must therefore include at least one access specifier.

Whether a property is hidden or overridden in a derived class, property look-up is always static. That is, the declared (compile-time) type of the variable used to identify an object determines the interpretation of its property identifiers. Hence, after the following code executes, reading or assigning a value to `MyObject.Value` invokes `Method1` or `Method2`, even though `MyObject` holds an instance of `TDescendant`. But you can cast `MyObject` to `TDescendant` to access the descendant class's properties and their access specifiers.

```
type
    TAncestor = class
```

```
      ...
      property Value: Integer read Method1 write Method2;
    end;

    TDescendant = class(TAncestor)
      ...
      property Value: Integer read Method3 write Method4;
    end;

 var MyObject: TAncestor;
      ...
     MyObject := TDescendant.Create;
```

Class Properties

Class properties can be accessed without an object reference. Class property accessors must themselves be declared as **class static** methods, or **class fields**. A class property is declared with the **class property** keywords. Class properties cannot be **published**, and cannot have **stored** or **default** value definitions.

You can introduce a block of class static fields within a class declaration by using the **class var** block declaration. All fields declared after **class var** have static storage attributes. A **class var** block is terminated by the following:

1. Another **class var** declaration
2. A procedure or function (i.e. method) declaration (see page 521) (including class procedures and class functions)
3. A property declaration (including class properties)
4. A constructor or destructor declaration (see page 521)
5. A visibility scope specifier (see page 514) (**public**, **private**, **protected**, **published**, **strict private**, and **strict protected**)

For example:

```
type
   TMyClass = class
     strict private
       class var         // Note fields must be declared as class fields
          FRed: Integer;
          FGreen: Integer;
          FBlue: Integer;
       public             // ends the class var block
          class property Red: Integer read FRed write FRed;
          class property Green: Integer read FGreen write FGreen;
          class property Blue: Integer read FBlue write FBlue;
   end;
```

You can access the above class properties with the code:

```
TMyClass.Red := 0;
TMyClass.Blue := 0;
TMyClass.Green := 0;
```

See Also

Classes and Objects (see page 514)

Fields (see page 519)

Methods (see page 521)

Nested Type Declarations (see page 546)

Class References (see page 539)

Exceptions (see page 541)

Operator Overloading (see page 548)

Class Helpers (see page 550)

3.1.3.1.5 Events

This topic describes the following material:

- Event properties and event handlers
- Triggering multiple event handlers
- Multicast events (.NET)

About Events

An event links an occurrence in the system with the code that responds to that occurrence. The occurrence triggers the execution of a procedure called an event handler. The event handler performs the tasks that are required in response to the occurrence. Events allow the behavior of a component to be customized at design-time or at runtime. To change the behavior of the component, replace the event handler with a custom event handler that will have the desired behavior.

Event Properties and Event Handlers

Components that are written in Delphi use properties to indicate the event handler that will be executed when the event occurs. By convention, the name of an event property begins with "On", and the property is implemented with a field rather than read/write methods. The value stored by the property is a method pointer, pointing to the event handler procedure.

In the following example, the `TObservedObject` class includes an `OnPing` event, of type `TPingEvent`. The `FOnPing` field is used to store the event handler. The event handler in this example, `TListener.Ping`, prints 'TListener has been pinged!'.

```
Program EventDemo;
{$APPTYPE CONSOLE}

type
    TPingEvent = procedure of object;
    TObservedObject = class
    private
        FPing: TPingEvent;
    public
        property OnPing: TPingEvent read FPing write FPing;
    end;

    TListener = class
        procedure Ping;
    end;

procedure TListener.Ping;
begin
    writeln('TListener has been pinged.');
end;

var
    observedObject: TObservedObject;
    listener: TListener;

begin
    observedObject := TObservedObject.Create;
    listener := TListener.Create;

    observedObject.OnPing := listener.Ping;

    observedObject.OnPing; // should output 'TListener has been pinged.'

    ReadLn; // pause console before closing
end.
```

Triggering Multiple Event Handlers

In Delphi for Win32, events can be assigned only a single event handler. If multiple event handlers must be executed in response to an event, the event handler assigned to the event must call any other event handlers. In the following code, a subclass of `TListener` called `TListenerSubclass` has its own event handler called `Ping2`. In this example, the `Ping2` event handler must explicitly call the `TListener.Ping` event handler in order to trigger it in response to the `OnPing` event.

```
Program EventDemo2;
{$APPTYPE CONSOLE}

type
    TPingEvent = procedure of object;
    TObservedObject = class
    private
        FPing: TPingEvent;
    public
        property OnPing: TPingEvent read FPing write FPing;
    end;

    TListener = class
        procedure Ping;
  end;

  TListenerSubclass = class (TListener)
    procedure Ping2;
    end;

procedure TListener.Ping;
begin
    writeln('TListener has been pinged.');
end;

procedure TListenerSubclass.Ping2;
begin
  self.Ping;
    writeln('TListenerSubclass has been pinged.');
end;

var
    observedObject: TObservedObject;
    listener: TListenerSubclass;

begin
    observedObject := TObservedObject.Create;
    listener := TListenerSubclass.Create;

    observedObject.OnPing := listener.Ping2;

    observedObject.OnPing; // should output 'TListener has been pinged.'
      // and then 'TListenerSubclass has been pinged.'

    ReadLn; // pause console before closing
end.
```

Multicast Events (.NET only)

On the .NET platform, Delphi enables multiple event handlers to be applied to the same event. A multicast event is an event that can trigger multiple event handlers. Using mulitcast events can reduce the effort required for maintenance of complex event systems, and improve the readability and flexibility of event handling.

Multicast events are declared using the **add** and **remove** keywords to indicate the field or methods that are used to add or remove event handlers for an event. The `Include()` and `Exclude()` standard procedures are used to include and exclude event handlers at runtime. The following code demonstrates the declaration of an event property that uses multicast events.

```
Program NETEvents;
```

```
{$APPTYPE CONSOLE}

type
    TPingEvent = procedure of object;
    TObservedObject = class
    private
        FPing: TPingEvent;
    public
        property OnPing: TPingEvent add FPing remove FPing;
    end;

    TListener = class
        procedure Ping;
  end;

  TListenerSubclass = class (TListener)
    procedure Ping2;
    end;

procedure TListener.Ping;
begin
    writeln('TListener has been pinged.');
end;

procedure TListenerSubclass.Ping2;
begin
    writeln('TListenerSubclass has been pinged.');
end;

var
    observedObject: TObservedObject;
    listener: TListener;
    listenerSubclass: TListenerSubclass;
    testEvent: TPingEvent;

begin
    observedObject := TObservedObject.Create;
    listener := TListener.Create;
    listenerSubclass := TListenerSubclass.Create;

    Include(observedObject.OnPing, listener.Ping);
    Include(observedObject.OnPing, listenerSubclass.Ping2);

    // testEvent := observedObject.OnPing;     // not allowed

    observedObject.FPing(); // should output 'TListener has been pinged.'

    ReadLn; // pause console before closing
end.
```

The `ObservedObject.OnPing` property declaration uses the **add** and **remove** keywords instead of **read** and **write**. The **add** and **remove** keywords indicate to the compiler that `OnClick` is a multicast event.

A property which represents a multicast event can not be read or written directly; it can only be accessed through the `Include()` and `Exclude()` standard procedures. To attempt to read, write or execute a property that has been declared with **add** and **remove** is an error at compile time. To execute this event directly, the sample code uses the `FPing` field rather than the `OnPing` property.

See Also

Classes and Objects (↗ see page 514)

Properties (↗ see page 530)

Methods (↗ see page 521)

Procedural types (see page 578)

Creating events

3.1.3.1.6 Class References

Sometimes operations are performed on a class itself, rather than on instances of a class (that is, objects). This happens, for example, when you call a constructor method using a class reference. You can always refer to a specific class using its name, but at times it is necessary to declare variables or parameters that take classes as values, and in these situations you need *class-reference types*.

This topic covers the following material:

- Class reference types
- Class operators

Class-Reference Types

A class-reference type, sometimes called a metaclass, is denoted by a construction of the form

`class of` *type*

where *type* is any class type. The identifier *type* itself denotes a value whose type is `class of` *type*. If `type1` is an ancestor of `type2`, then `class of type2` is assignment-compatible with class of `type1`. Thus

```
type TClass = class of TObject;
var AnyObj: TClass;
```

declares a variable called `AnyObj` that can hold a reference to any class. (The definition of a class-reference type cannot occur directly in a variable declaration or parameter list.) You can assign the value **nil** to a variable of any class-reference type.

To see how class-reference types are used, look at the declaration of the constructor for TCollection (in the `Classes` unit):

```
type TCollectionItemClass = class of TCollectionItem;
     ...
constructor Create(ItemClass: TCollectionItemClass);
```

This declaration says that to create a TCollection instance object, you must pass to the constructor the name of a class descending from TCollectionItem.

Class-reference types are useful when you want to invoke a class method or virtual constructor on a class or object whose actual type is unknown at compile time.

Constructors and Class References

A constructor can be called using a variable of a class-reference type. This allows construction of objects whose type isn't known at compile time. For example,

```
type TControlClass = class of TControl;

function CreateControl(ControlClass: TControlClass;
const ControlName: string; X, Y, W, H: Integer): TControl;
begin
     Result := ControlClass.Create(MainForm);
     with Result do
      begin
        Parent := MainForm;
        Name := ControlName;
        SetBounds(X, Y, W, H);
        Visible := True;
      end;
end;
```

The `CreateControl` function requires a class-reference parameter to tell it what kind of control to create. It uses this parameter to call the class's constructor. Because class-type identifiers denote class-reference values, a call to `CreateControl` can specify the identifier of the class to create an instance of. For example,

```
CreateControl(TEdit, 'Edit1', 10, 10, 100, 20);
```

Constructors called using class references are usually virtual. The constructor implementation activated by the call depends on the runtime type of the class reference.

Class Operators

Class methods (see page 521) operate on class references. Every class inherits two class methods from TObject, called ClassType and ClassParent. These methods return, respectively, a reference to the class of an object and to an object's immediate ancestor class. Both methods return a value of type TClass (where `TClass = class of TObject`), which can be cast to a more specific type. Every class also inherits a method called InheritsFrom that tests whether the object where it is called descends from a specified class. These methods are used by the **is** and **as** operators, and it is seldom necessary to call them directly.

The is Operator

The **is** operator, which performs dynamic type checking, is used to verify the actual runtime class of an object. The expression

*object*__is__*class*

returns **True** if *object* is an instance of the class denoted by class or one of its descendants, and **False** otherwise. (If object is **nil**, the result is **False**.) If the declared type of object is unrelated to class - that is, if the types are distinct and one is not an ancestor of the othera compilation error results. For example,

```
if ActiveControl is TEdit then TEdit(ActiveControl).SelectAll;
```

This statement casts a variable to `TEdit` after first verifying that the object it references is an instance of `TEdit` or one of its descendants.

The as Operator

The **as** operator performs checked typecasts. The expression

*object*__as__*class*

returns a reference to the same object as object, but with the type given by class. At runtime, object must be an instance of the class denoted by class or one of its descendants, or be **nil**; otherwise an exception is raised. If the declared type of object is unrelated to class - that is, if the types are distinct and one is not an ancestor of the other - a compilation error results. For example,

```
with Sender as TButton do
 begin
  Caption := '&Ok';
  OnClick := OkClick;
 end;
```

The rules of operator precedence often require **as** typecasts to be enclosed in parentheses. For example,

```
(Sender as TButton).Caption := '&Ok';
```

See Also

Classes and Objects (see page 514)

Fields (see page 519)

Methods (see page 521)

Properties (see page 530)

Nested Type Declarations (see page 546)

Exceptions (see page 541)

Operator Overloading (see page 548)

Class Helpers (see page 550)

3.1.3.1.7 Exceptions

This topic covers the following material:

- A conceptual overview of exceptions and exception handling
- Declaring exception types
- Raising and handling exceptions

About Exceptions

An exception is raised when an error or other event interrupts normal execution of a program. The exception transfers control to an exception handler, which allows you to separate normal program logic from error-handling. Because exceptions are objects, they can be grouped into hierarchies using inheritance, and new exceptions can be introduced without affecting existing code. An exception can carry information, such as an error message, from the point where it is raised to the point where it is handled.

When an application uses the `SysUtils` unit, most runtime errors are automatically converted into exceptions. Many errors that would otherwise terminate an application - such as insufficient memory, division by zero, and general protection faults - can be caught and handled.

When To Use Exceptions

Exceptions provide an elegant way to trap runtime errors without halting the program and without awkward conditional statements. The requirements imposed by exception handling semantics impose a code/data size and runtime performance penalty. While it is possible to raise exceptions for almost any reason, and to protect almost any block of code by wrapping it in a `try...except` or `try...finally` statement, in practice these tools are best reserved for special situations.

Exception handling is appropriate for errors whose chances of occurring are low or difficult to assess, but whose consequences are likely to be catastrophic (such as crashing the application); for error conditions that are complicated or difficult to test for in `if...then` statements; and when you need to respond to exceptions raised by the operating system or by routines whose source code you don't control. Exceptions are commonly used for hardware, memory, I/O, and operating-system errors.

Conditional statements are often the best way to test for errors. For example, suppose you want to make sure that a file exists before trying to open it. You could do it this way:

```
try
    AssignFile(F, FileName);
    Reset(F);     // raises an EInOutError exception if file is not found
except
    on Exception do ...
end;
```

But you could also avoid the overhead of exception handling by using

```
if FileExists(FileName) then    // returns False if file is not found; raises no exception
begin
    AssignFile(F, FileName);
    Reset(F);
end;
```

Assertions provide another way of testing a Boolean condition anywhere in your source code. When an `Assert` statement fails, the program either halts with a runtime error or (if it uses the `SysUtils` unit) raises an EAssertionFailed exception. Assertions should be used only to test for conditions that you do not expect to occur.

Declaring Exception Types

Exception types are declared just like other classes. In fact, it is possible to use an instance of any class as an exception, but it is recommended that exceptions be derived from the Exception class defined in `SysUtils`.

You can group exceptions into families using inheritance. For example, the following declarations in `SysUtils` define a family of exception types for math errors.

```
type
   EMathError = class(Exception);
   EInvalidOp = class(EMathError);
   EZeroDivide = class(EMathError);
   EOverflow = class(EMathError);
   EUnderflow = class(EMathError);
```

Given these declarations, you can define a single EMathError exception handler that also handles EInvalidOp, EZeroDivide, EOverflow, and EUnderflow.

Exception classes sometimes define fields, methods, or properties that convey additional information about the error. For example,

```
type EInOutError = class(Exception)
        ErrorCode: Integer;
      end;
```

Raising and Handling Exceptions

To raise an exception object, use an instance of the exception class with a **raise** statement. For example,

```
raise EMathError.Create;
```

In general, the form of a **raise** statement is

raise_object_**at**_address_

where object and at address are both optional. When an address is specified, it can be any expression that evaluates to a pointer type, but is usually a pointer to a procedure or function. For example:

```
raise Exception.Create('Missing parameter') at @MyFunction;
```

Use this option to raise the exception from an earlier point in the stack than the one where the error actually occurred.

When an exception is raised - that is, referenced in a **raise** statement - it is governed by special exception-handling logic. A **raise** statement never returns control in the normal way. Instead, it transfers control to the innermost exception handler that can handle exceptions of the given class. (The innermost handler is the one whose `try...except` block was most recently entered but has not yet exited.)

For example, the function below converts a string to an integer, raising an ERangeError exception if the resulting value is outside a specified range.

```
function StrToIntRange(const S: string; Min, Max: Longint): Longint;
begin
    Result := StrToInt(S);   // StrToInt is declared in SysUtils
    if (Result < Min) or (Result > Max) then
       raise ERangeError.CreateFmt('%d is not within the valid range of %d..%d', [Result, Min,
Max]);
end;
```

Notice the *CreateFmt* method called in the **raise** statement. Exception and its descendants have special constructors that provide alternative ways to create exception messages and context IDs.

A raised exception is destroyed automatically after it is handled. Never attempt to destroy a raised exception manually.

Note: Raising an exception in the initialization section of a unit may not produce the intended result. Normal exception support comes from the `SysUtils` unit, which must be initialized before such support is available. If an exception occurs during initialization, all initialized units - including `SysUtils` - are finalized and the exception is re-raised. Then the exception is caught and handled, usually by interrupting the program. Similarly, raising an exception in the finalization section of a unit may not lead to the intended result if `SysUtils` has already been finalized when the exception has been raised.

Try...except Statements

Exceptions are handled within `try...except` statements. For example,

```
try
   X := Y/Z;
   except
     on EZeroDivide do HandleZeroDivide;
end;
```

This statement attempts to divide `Y` by `Z`, but calls a routine named `HandleZeroDivide` if an EZeroDivide exception is raised.

The syntax of a `try...except` statement is

try *statements***except***exceptionBlock***end**

where statements is a sequence of statements (delimited by semicolons) and *exceptionBlock* is either

- another sequence of statements or
- a sequence of exception handlers, optionally followed by

else*statements*

An exception handler has the form

on*identifier: type***do***statement*

where *identifier:* is optional (if included, identifier can be any valid identifier), type is a type used to represent exceptions, and *statement* is any statement.

A `try...except` statement executes the statements in the initial statements list. If no exceptions are raised, the exception block (*exceptionBlock*) is ignored and control passes to the next part of the program.

If an exception is raised during execution of the initial statements list, either by a **raise** statement in the statements list or by a procedure or function called from the statements list, an attempt is made to 'handle' the exception:

- If any of the handlers in the exception block matches the exception, control passes to the first such handler. An exception handler 'matches' an exception just in case the type in the handler is the class of the exception or an ancestor of that class.
- If no such handler is found, control passes to the statement in the **else** clause, if there is one.
- If the exception block is just a sequence of statements without any exception handlers, control passes to the first statement in the list.

If none of the conditions above is satisfied, the search continues in the exception block of the next-most-recently entered `try...except` statement that has not yet exited. If no appropriate handler, **else** clause, or statement list is found there, the search propagates to the next-most-recently entered `try...except` statement, and so forth. If the outermost `try...except` statement is reached and the exception is still not handled, the program terminates.

When an exception is handled, the stack is traced back to the procedure or function containing the `try...except` statement where the handling occurs, and control is transferred to the executed exception handler, **else** clause, or statement list. This process discards all procedure and function calls that occurred after entering the `try...except` statement where the exception is handled. The exception object is then automatically destroyed through a call to its `Destroy` destructor and control is passed to the statement following the `try...except` statement. (If a call to the `Exit`, `Break`, or `Continue` standard procedure causes control to leave the exception handler, the exception object is still automatically destroyed.)

In the example below, the first exception handler handles division-by-zero exceptions, the second one handles overflow exceptions, and the final one handles all other math exceptions. EMathError appears last in the exception block because it is the ancestor of the other two exception classes; if it appeared first, the other two handlers would never be invoked.

```
try
  ...
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
end;
```

An exception handler can specify an identifier before the name of the exception class. This declares the identifier to represent the exception object during execution of the statement that follows `on...do`. The scope of the identifier is limited to that statement. For example,

```
try
  ...
except
  on E: Exception do ErrorDialog(E.Message, E.HelpContext);
end;
```

If the exception block specifies an **else** clause, the **else** clause handles any exceptions that aren't handled by the block's exception handlers. For example,

```
try
  ...
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
else
  HandleAllOthers;
end;
```

Here, the **else** clause handles any exception that isn't an EMathError.

An exception block that contains no exception handlers, but instead consists only of a list of statements, handles all exceptions. For example,

```
try
   ...
except
   HandleException;
end;
```

Here, the `HandleException` routine handles any exception that occurs as a result of executing the statements between **try** and **except**.

Re-raising Exceptions

When the reserved word **raise** occurs in an exception block without an object reference following it, it raises whatever exception is handled by the block. This allows an exception handler to respond to an error in a limited way and then re-raise the exception. Re-raising is useful when a procedure or function has to clean up after an exception occurs but cannot fully handle the exception.

For example, the `GetFileList` function allocates a `TStringList` object and fills it with file names matching a specified search path:

```
function GetFileList(const Path: string): TStringList;
var
  I: Integer;
  SearchRec: TSearchRec;
begin
  Result := TStringList.Create;
  try
    I := FindFirst(Path, 0, SearchRec);
    while I = 0 do
      begin
          Result.Add(SearchRec.Name);
          I := FindNext(SearchRec);
      end;
  except
      Result.Free;
```

```
        raise;
    end;
  end;
```

GetFileList creates a TStringList object, then uses the FindFirst and FindNext functions (defined in SysUtils) to initialize it. If the initialization fails - for example because the search path is invalid, or because there is not enough memory to fill in the string list - GetFileList needs to dispose of the new string list, since the caller does not yet know of its existence. For this reason, initialization of the string list is performed in a try...except statement. If an exception occurs, the statement's exception block disposes of the string list, then re-raises the exception.

Nested Exceptions

Code executed in an exception handler can itself raise and handle exceptions. As long as these exceptions are also handled within the exception handler, they do not affect the original exception. However, once an exception raised in an exception handler propagates beyond that handler, the original exception is lost. This is illustrated by the Tan function below.

```
type
   ETrigError = class(EMathError);
   function Tan(X: Extended): Extended;
   begin
      try
        Result := Sin(X) / Cos(X);
      except
        on EMathError do
        raise ETrigError.Create('Invalid argument to Tan');
      end;
   end;
```

If an EMathError exception occurs during execution of Tan, the exception handler raises an ETrigError. Since Tan does not provide a handler for ETrigError, the exception propagates beyond the original exception handler, causing the EMathError exception to be destroyed. To the caller, it appears as if the Tan function has raised an ETrigError exception.

Try...finally Statements

Sometimes you want to ensure that specific parts of an operation are completed, whether or not the operation is interrupted by an exception. For example, when a routine acquires control of a resource, it is often important that the resource be released, regardless of whether the routine terminates normally. In these situations, you can use a try...finally statement.

The following example shows how code that opens and processes a file can ensure that the file is ultimately closed, even if an error occurs during execution.

```
Reset(F);
try
   ... // process file F
finally
   CloseFile(F);
end;
```

The syntax of a try...finally statement is

try*statementList1***finally***statementList2***end**

where each *statementList* is a sequence of statements delimited by semicolons. The try...finally statement executes the statements in *statementList1* (the **try** clause). If *statementList1* finishes without raising exceptions, *statementList2* (the **finally** clause) is executed. If an exception is raised during execution of *statementList1*, control is transferred to *statementList2*; once *statementList2* finishes executing, the exception is re-raised. If a call to the Exit, Break, or Continue procedure causes control to leave *statementList1*, *statementList2* is automatically executed. Thus the **finally** clause is always executed, regardless of how the **try** clause terminates.

If an exception is raised but not handled in the **finally** clause, that exception is propagated out of the try...finally statement, and any exception already raised in the **try** clause is lost. The **finally** clause should therefore handle all locally raised exceptions, so as not to disturb propagation of other exceptions.

Standard Exception Classes and Routines

The `SysUtils` and `System` units declare several standard routines for handling exceptions, including ExceptObject, `ExceptAddr`, and ShowException. `SysUtils`, `System` and other units also include dozens of exception classes, all of which (aside from `OutlineError`) derive from Exception.

The Exception class has properties called Message and `HelpContext` that can be used to pass an error description and a context ID for context-sensitive online documentation. It also defines various constructor methods that allow you to specify the description and context ID in different ways.

See Also

Classes and Objects (see page 514)

Fields (see page 519)

Methods (see page 521)

Properties (see page 530)

Nested Type Declarations (see page 546)

Class References (see page 539)

Operator Overloading (see page 548)

Class Helpers (see page 550)

3.1.3.1.8 Nested Type Declarations

Type declarations can be nested within class declarations. Nested types are used throughout the .NET framework, and throughout object-oriented programming in general. They present a way to keep conceptually related types together, and to avoid name collisions. The same syntax for declaring nested types may be used with the Win32 Delphi compiler.

Declaring Nested Types

The *nestedTypeDeclaration* follows the type declaration syntax defined in Declaring Types (see page 586).

```
type
className = class [abstract | sealed] (ancestorType)
    memberList

    type
       nestedTypeDeclaration

    memberList
end;
```

Nested type declarations are terminated by the first occurance of a non-identifier token, for example, **procedure**, **class**, **type**, and all visibility scope specifiers.

The normal accessibility rules apply to nested types and their containing types. A nested type can access an instance variable (field, property, or method) of its container class, but it must have an object reference to do so. A nested type can access class fields, class properties, and class static methods without an object reference, but the normal Delphi visibility rules apply.

Nested types do not increase the size of the containing class. Creating an instance of the containing class does not also create an instance of a nested type. Nested types are associated with their containing classes only by the context of their declaration.

Declaring and Accessing Nested Classes

The following example demonstrates how to declare and access fields and methods of a nested class.

```
type
      TOuterClass = class
```

```
         strict private
            myField: Integer;

         public
            type
               TInnerClass = class
                public
                  myInnerField: Integer;
                  procedure innerProc;
               end;

           procedure outerProc;
         end;
```

To implement the `innerProc` method of the inner class, you must qualify its name with the name of the outer class. For example

```
procedure TOuterClass.TInnerClass.innerProc;
begin
   ...
end;
```

To access the members of the nested type, use dotted notation as with regular class member access. For example

```
var
   x: TOuterClass;
   y: TOuterClass.TInnerClass;

begin
   x := TOuterClass.Create;
   x.outerProc;
   ...
   y := TOuterClass.TInnerClass.Create;
   y.innerProc;
```

Nested Constants

Constants can be declared in class types in the same manner as nested type sections. Constant sections are terminated by the same tokens as nested type sections, specifically, reserved words or visibility specifiers. Typed constants are not supported, so you cannot declare nested constants of value types, such as Currency, or TDateTime.

Nested constants can be of any simple type: ordinal, ordinal subranges, enums, strings, and real types.

The following code demonstrates the declaration of nested constants:

```
type
   TMyClass = class
        const
             x = 12;
             y = TMyClass.x + 23;
        procedure Hello;
        private
             const
                 s = 'A string constant';
    end;

begin
   writeln(TMyClass.y);   // Writes the value of y, 35.
end.
```

See Also

Classes and Objects (see page 514)

Fields (see page 519)

Methods (see page 521)

Properties (see page 530)

Class References (see page 539)

Exceptions (see page 541)

Operator Overloading (see page 548)

Class Helpers (see page 550)

3.1.3.1.9 Operator Overloading

This topic describes Delphi's operator methods and how to overload them.

About Operator Overloading

Delphi for .NET and Delphi for Win32 allow certain functions, or "operators" to be overloaded within record declarations. Delphi for .NET also allows overloading within class declarations. The name of the operator function maps to a symbolic representation in source code. For example, the `Add` operator maps to the **+** symbol. The compiler generates a call to the appropriate overload, matching the context (i.e. the return type, and type of parameters used in the call), to the signature of the operator function. The following table shows the Delphi operators that can be overloaded:

Operator	Category	Declaration Signature	Symbol Mapping
`Implicit`	Conversion	`Implicit(a : type) : resultType;`	implicit typecast
`Explicit`	Conversion	`Explicit(a: type) : resultType;`	explicit typecast
`Negative`	Unary	`Negative(a: type) : resultType;`	-
`Positive`	Unary	`Positive(a: type): resultType;`	+
`Inc`	Unary	`Inc(a: type) : resultType;`	`Inc`
`Dec`	Unary	`Dec(a: type): resultType`	`Dec`
`LogicalNot`	Unary	`LogicalNot(a: type): resultType;`	**not**
`BitwiseNot`	Unary	`BitwiseNot(a: type): resultType;`	**not**
`Trunc`	Unary	`Trunc(a: type): resultType;`	`Trunc`
`Round`	Unary	`Round(a: type): resultType;`	`Round`
`Equal`	Comparison	`Equal(a: type; b: type) : Boolean;`	=
`NotEqual`	Comparison	`NotEqual(a: type; b: type): Boolean;`	<>
`GreaterThan`	Comparison	`GreaterThan(a: type; b: type) Boolean;`	>
`GreaterThanOrEqual`	Comparison	`GreaterThanOrEqual(a: type; b: type): resultType;`	>=
`LessThan`	Comparison	`LessThan(a: type; b: type): resultType;`	<
`LessThanOrEqual`	Comparison	`LessThanOrEqual(a: type; b: type): resultType;`	<=
`Add`	Binary	`Add(a: type; b: type): resultType;`	+
`Subtract`	Binary	`Subtract(a: type; b: type) : resultType;`	-
`Multiply`	Binary	`Multiply(a: type; b: type) : resultType;`	*
`Divide`	Binary	`Divide(a: type; b: type) : resultType;`	/
`IntDivide`	Binary	`IntDivide(a: type; b: type): resultType;`	**div**
`Modulus`	Binary	`Modulus(a: type; b: type): resultType;`	**mod**

`LeftShift`	Binary	`LeftShift(a: type; b: type): resultType;`	**shl**
`RightShift`	Binary	`RightShift(a: type; b: type): resultType;`	**shr**
`LogicalAnd`	Binary	`LogicalAnd(a: type; b: type): resultType;`	**and**
`LogicalOr`	Binary	`LogicalOr(a: type; b: type): resultType;`	**or**
`LogicalXor`	Binary	`LogicalXor(a: type; b: type): resultType;`	**xor**
`BitwiseAnd`	Binary	`BitwiseAnd(a: type; b: type): resultType;`	**and**
`BitwiseOr`	Binary	`BitwiseOr(a: type; b: type): resultType;`	**or**
`BitwiseXor`	Binary	`BitwiseXor(a: type; b: type): resultType;`	**xor**

No operators other than those listed in the table may be defined on a class or record.

Overloaded operator methods cannot be referred to by name in source code. To access a specific operator method of a specific class or record, you must use explicit typecasts on all of the operands. Operator identifiers are not included in the class or record's list of members.

No assumptions are made regarding the distributive or commutative properties of the operation. For binary operators, the first parameter is always the left operand, and the second parameter is always the right operand. Associativity is assumed to be left-to-right in the absence of explicit parentheses.

Resolution of operator methods is done over the union of accessible operators of the types used in the operation (note this includes inherited operators). For an operation involving two different types A and B, if type A has an implicit conversion to B, and B has an implicit conversion to A, an ambiguity will occur. Implicit conversions should be provided only where absolutely necessary, and reflexivity should be avoided. It is best to let type B implicitly convert itself to type A, and let type A have no knowledge of type B (or vice versa).

As a general rule, operators should not modify their operands. Instead, return a new value, constructed by performing the operation on the parameters.

Overloaded operators are used most often in records (i.e. value types). Very few classes in the .NET framework have overloaded operators, but most value types do.

Declaring Operator Overloads

Operator overloads are declared within classes or records, with the following syntax:

```
type
   typeName = [class | record]
       class operator conversionOp(a: type): resultType;
       class operator unaryOp(a: type): resultType;
       class operator comparisonOp(a: type; b: type): Boolean;
       class operator binaryOp(a: type; b: type): resultType;
   end;
```

Implementation of overloaded operators must also include the **class operator** syntax:

```
class operator typeName.conversionOp(a: type): resultType;
class operator typeName.unaryOp(a: type): resultType;
class operator typeName.comparisonOp(a: type; b: type): Boolean;
class operator typeName.binaryOp(a: type; b: type): resultType;
```

The following are some examples of overloaded operators:

```
type
   TMyClass = class
     class operator Add(a, b: TMyClass): TMyClass;      // Addition of two operands of type
TMyClass
     class operator Subtract(a, b: TMyClass): TMyclass; // Subtraction of type TMyClass
     class operator Implicit(a: Integer): TMyClass;     // Implicit conversion of an Integer
```

```
to type TMyClass
     class operator Implicit(a: TMyClass): Integer;     // Implicit conversion of TMyClass to
Integer
     class operator Explicit(a: Double): TMyClass;      // Explicit conversion of a Double to
TMyClass
   end;

// Example implementation of Add
class operator TMyClass.Add(a, b: TMyClass): TMyClass;
begin
   // ...
end;

var
x, y: TMyClass;
begin
   x := 12;      // Implicit conversion from an Integer
   y := x + x;   // Calls TMyClass.Add(a, b: TMyClass): TMyClass
   b := b + 100; // Calls TMyClass.Add(b, TMyClass.Implicit(100))
end;
```

See Also

Classes and Objects (see page 514)

Fields (see page 519)

Methods (see page 521)

Properties (see page 530)

Nested Type Declarations (see page 546)

Class Helpers (see page 550)

Class References (see page 539)

Exceptions (see page 541)

3.1.3.1.10 Class Helpers

This topic describes the syntax of class helper declarations.

About Class Helpers

A class helper is a type that - when associated with another class - introduces additional method names and properties which may be used in the context of the associated class (or its descendants). Class helpers are a way to extend a class without using inheritance. A class helper simply introduces a wider scope for the compiler to use when resolving identifiers. When you declare a class helper, you state the helper name, and the name of the class you are going to extend with the helper. You can use the class helper any place where you can legally use the extended class. The compiler's resolution scope then becomes the original class, plus the class helper.

Class helpers provide a way to extend a class, but they should not be viewed as a design tool to be used when developing new code. They should be used solely for their intended purpose, which is language and platform RTL binding.

Class Helper Syntax

The syntax for declaring a class helper is:

```
type
   identifierName = class helper [(ancestor list)] for classTypeIdentifierName
     memberList
   end;
```

The *ancestor list* is optional.

A class helper type may not declare instance data, but class fields (see page 519) are allowed.

The visibility scope rules and *memberList* syntax are identical to that of ordinary class types.

You can define and associate multiple class helpers with a single class type. However, only zero or one class helper applies in any specific location in source code. The class helper defined in the nearest scope will apply. Class helper scope is determined in the normal Delphi fashion (i.e. right to left in the unit's **uses** clause).

Using Class Helpers

The following code demonstrates the declaration of a class helper:

```
type
   TMyClass = class
      procedure MyProc;
      function  MyFunc: Integer;
   end;

   ...

   procedure TMyClass.MyProc;
   var X: Integer;
   begin
      X := MyFunc;
   end;

   function TMyClass.MyFunc: Integer;
   begin
       ...
   end;

...

type
   TMyClassHelper = class helper for TMyClass
     procedure HelloWorld;
     function MyFunc: Integer;
   end;

   ...

   procedure TMyClassHelper.HelloWorld;
   begin
      writeln(Self.ClassName); // Self refers to TMyClass type, not TMyClassHelper
   end;

   function TMyClassHelper.MyFunc: Integer;
   begin
     ...
   end;

...

var
  X: TMyClass;
begin
  X := TMyClass.Create;
  X.MyProc;    // Calls TMyClass.MyProc
  X.HelloWorld; // Calls TMyClassHelper.HelloWorld
  X.MyFunc;    // Calls TMyClassHelper.MyFunc
```

Note that the class helper function `MyFunc` is called, since the class helper takes precedence over the actual class type.

See Also

Classes and Objects (see page 514)

Fields (see page 519)

Methods (see page 521)

Properties (see page 530)

Nested Type Declarations (see page 546)

Operator Overloading (see page 548)

Class References (see page 539)

Exceptions (see page 541)

3.1.3.2 Data Types, Variables, and Constants

This section describes the fundamental data types of the Delphi language.

Topics

Name	Description
Data Types, Variables, and Constants (see page 553)	This topic presents a high-level overview of Delphi data types.
Simple Types (see page 554)	Simple types - which include ordinal types and real types - define ordered sets of values. The ordinal types covered in this topic are: • Integer types • Character types • Boolean types • Enumerated types • Real (floating point) types
String Types (see page 561)	This topic describes the string data types available in the Delphi language. The following types are covered: • Short strings. • Long strings. • Wide (Unicode) strings.
Structured Types (see page 566)	Instances of a structured type hold more than one value. Structured types include sets, arrays, records, and files as well as class, class-reference, and interface types. Except for sets, which hold ordinal values only, structured types can contain other structured types; a type can have unlimited levels of structuring. **Note:** Typed and untyped file types are not supported with the .NET framework. By default, the values in a structured type are aligned on word or double-word boundaries for faster access. When you declare a structured type, you can include the reserved word **packed** to implement compressed data storage. For example,... more (see page 566)
Pointers and Pointer Types (see page 575)	A pointer is a variable that denotes a memory address. When a pointer holds the address of another variable, we say that it points to the location of that variable in memory or to the data stored there. In the case of an array or other structured type, a pointer holds the address of the first element in the structure. If that address is already taken, then the pointer holds the address to the first element. Pointers are typed to indicate the kind of data stored at the addresses they hold. The general-purpose **Pointer** type can represent a pointer to... more (see page 575)
Procedural Types (see page 578)	Procedural types allow you to treat procedures and functions as values that can be assigned to variables or passed to other procedures and functions.
Variant Types (see page 580)	This topic discusses the use of variant data types.

Type Compatibility and Identity (↗ see page 583)	To understand which operations can be performed on which expressions, we need to distinguish several kinds of compatibility among types and values. These include: • Type identity • Type compatibility • Assignment compatibility
Declaring Types (↗ see page 586)	This topic describes the syntax of Delphi type declarations.
Variables (↗ see page 587)	A variable is an identifier whose value can change at runtime. Put differently, a variable is a name for a location in memory; you can use the name to read or write to the memory location. Variables are like containers for data, and, because they are typed, they tell the compiler how to interpret the data they hold.
Declared Constants (↗ see page 589)	Several different language constructions are referred to as 'constants'. There are numeric constants (also called numerals) like 17, and string constants (also called character strings or string literals) like 'Hello world!'. Every enumerated type defines constants that represent the values of that type. There are predefined constants like **True**, **False**, and **nil**. Finally, there are constants that, like variables, are created individually by declaration. Declared constants are either *true constants* or *typed constants*. These two kinds of constant are superficially similar, but they are governed by different rules and used for different purposes.

3.1.3.2.1 Data Types, Variables, and Constants

This topic presents a high-level overview of Delphi data types.

About Types

A type is essentially a name for a kind of data. When you declare a variable you must specify its type, which determines the set of values the variable can hold and the operations that can be performed on it. Every expression returns data of a particular type, as does every function. Most functions and procedures require parameters of specific types.

The Delphi language is a 'strongly typed' language, which means that it distinguishes a variety of data types and does not always allow you to substitute one type for another. This is usually beneficial because it lets the compiler treat data intelligently and validate your code more thoroughly, preventing hard-to-diagnose runtime errors. When you need greater flexibility, however, there are mechanisms to circumvent strong typing. These include typecasting, pointers, Variants, Variant parts in records, and absolute addressing of variables.

There are several ways to categorize Delphi data types:

- Some types are predefined (or built-in); the compiler recognizes these automatically, without the need for a declaration. Almost all of the types documented in this language reference are predefined. Other types are created by declaration; these include user-defined types and the types defined in the product libraries.
- Types can be classified as either fundamental or generic. The range and format of a fundamental type is the same in all implementations of the Delphi language, regardless of the underlying CPU and operating system. The range and format of a generic type is platform-specific and could vary across different implementations. Most predefined types are fundamental, but a handful of integer, character, string, and pointer types are generic. It's a good idea to use generic types when possible, since they provide optimal performance and portability. However, changes in storage format from one implementation of a generic type to the next could cause compatibility problems - for example, if you are streaming content to a file as raw, binary data, without type and versioning information.
- Types can be classified as simple, string, structured, pointer, procedural, or variant. In addition, type identifiers themselves can be regarded as belonging to a special 'type' because they can be passed as parameters to certain functions (such as **High**, **Low**, and **SizeOf**).
- Types can be parameterized , or

The outline below shows the taxonomy of Delphi data types.

```
simple
  ordinal
   integer
   character
   Boolean
   enumerated
   subrange
  real
 string
 structured
  set
  array
  record
  file
  class
  class reference
  interface
 pointer
 procedural
 Variant
 type identifier
```

The standard function `SizeOf` operates on all variables and type identifiers. It returns an integer representing the amount of memory (in bytes) required to store data of the specified type. For example, `SizeOf(Longint)` returns 4, since a **Longint** variable uses four bytes of memory.

Type declarations are illustrated in the topics that follow. For general information about type declarations, see Declaring types (see page 586).

See Also

Simple Types (see page 554)

String Types (see page 561)

Structured Types (see page 566)

Pointers and Pointer Types (see page 575)

Procedural Types (see page 578)

Variant Types (see page 580)

Type Compatibility and Identity (see page 583)

Declaring Types (see page 586)

Variables (see page 587)

Declared Constants (see page 589)

3.1.3.2.2 Simple Types

Simple types - which include ordinal types and real types - define ordered sets of values.

The ordinal types covered in this topic are:

- Integer types
- Character types
- Boolean types
- Enumerated types
- Real (floating point) types

Ordinal Types

Ordinal types include integer, character, Boolean, enumerated, and subrange types. An ordinal type defines an ordered set of values in which each value except the first has a unique predecessor and each value except the last has a unique successor. Further, each value has an ordinality which determines the ordering of the type. In most cases, if a value has ordinality *n*, its predecessor has ordinality *n-1* and its successor has ordinality *n+1*.

- For integer types, the ordinality of a value is the value itself.
- Subrange types maintain the ordinalities of their base types.
- For other ordinal types, by default the first value has ordinality 0, the next value has ordinality 1, and so forth. The declaration of an enumerated type can explicitly override this default.

Several predefined functions operate on ordinal values and type identifiers. The most important of them are summarized below.

Function	Parameter	Return value	Remarks
`Ord`	ordinal expression	ordinality of expression's value	Does not take Int64 arguments.
`Pred`	ordinal expression	predecessor of expression's value	
`Succ`	ordinal expression	successor of expression's value	
`High`	ordinal type identifier or variable of ordinal type	highest value in type	Also operates on short-string types and arrays.
`Low`	ordinal type identifier or variable of ordinal type	lowest value in type	Also operates on short-string types and arrays.

For example, `High(Byte)` returns 255 because the highest value of type **Byte** is 255, and `Succ(2)` returns 3 because 3 is the successor of 2.

The standard procedures `Inc` and `Dec` increment and decrement the value of an ordinal variable. For example, `Inc(I)` is equivalent to `I := Succ(I)` and, if `I` is an integer variable, to `I := I + 1`.

Integer Types

An integer type represents a subset of the whole numbers. The generic integer types are **Integer** and **Cardinal**; use these whenever possible, since they result in the best performance for the underlying CPU and operating system. The table below gives their ranges and storage formats for the Delphi compiler.

Generic integer types

Type	Range	Format	.NET Type Mapping
Integer	-2147483648..2147483647	signed 32-bit	Int32
Cardinal	0..4294967295	unsigned 32-bit	UInt32

Fundamental integer types include **Shortint**, **Smallint**, **Longint**, **Int64**, **Byte**, **Word**, **Longword, and UInt64**.

Fundamental integer types

Type	Range	Format	.NET Type Mapping
Shortint	-128..127	signed 8-bit	SByte
Smallint	-32768..32767	signed 16-bit	Int16
Longint	-2147483648..2147483647	signed 32-bit	Int32
Int64	-2^63..2^63-1	signed 64-bit	Int64
Byte	0..255	unsigned 8-bit	Byte
Word	0..65535	unsigned 16-bit	UInt16

Longword	0..4294967295	unsigned 32-bit	UInt32
UInt64	0..2^64–1	unsigned 64-bit	UInt64

In general, arithmetic operations on integers return a value of type **Integer**, which is equivalent to the 32-bit **Longint**. Operations return a value of type **Int64** only when performed on one or more **Int64** operand. Hence the following code produces incorrect results.

```
var
  I: Integer;
  J: Int64;
   ...

  I := High(Integer);
  J := I + 1;
```

To get an **Int64** return value in this situation, cast `I` as **Int64**:

```
...
J := Int64(I) + 1;
```

For more information, see Arithmetic operators (see page 720).

Note: Some standard routines that take integer arguments truncate Int64

values to 32 bits. However, the `High`, `Low`, `Succ`, `Pred`, `Inc`, `Dec`, `IntToStr`, and `IntToHex` routines fully support **Int64** arguments. Also, the `Round`, `Trunc`, `StrToInt64`, and `StrToInt64Def` functions return **Int64** values. A few routines cannot take **Int64** values at all. When you increment the last value or decrement the first value of an integer type, the result wraps around the beginning or end of the range. For example, the **Shortint** type has the range 128..127; hence, after execution of the code

```
var I: Shortint;
  ...
  I := High(Shortint);
  I := I + 1;
```

the value of `I` is 128. If compiler range-checking is enabled, however, this code generates a runtime error.

Character Types

The fundamental character types are **AnsiChar** and **WideChar**. **AnsiChar** values are byte-sized (8-bit) characters ordered according to the locale character set which is possibly multibyte. **AnsiChar** was originally modeled after the ANSI character set (thus its name) but has now been broadened to refer to the current locale character set.

WideChar characters use more than one byte to represent every character. In the current implementations, **WideChar** is word-sized (16-bit) characters ordered according to the Unicode character set (note that it could be longer in future implementations). The first 256 Unicode characters correspond to the ANSI characters.

The generic character type is **Char**, which is equivalent to **AnsiChar** on Win32, and to Char on the .NET platform. Because the implementation of **Char** is subject to change, it's a good idea to use the standard function `SizeOf` rather than a hard-coded constant when writing programs that may need to handle characters of different sizes.

Note: The WideChar

type also maps to Char on the .NET platform. A string constant of length 1, such as 'A', can denote a character value. The predefined function `Chr` returns the character value for any integer in the range of **AnsiChar** or **WideChar**; for example, `Chr(65)` returns the letter A.

Character values, like integers, wrap around when decremented or incremented past the beginning or end of their range (unless range-checking is enabled). For example, after execution of the code

```
var
```

```
  Letter: Char;
  I: Integer;
begin
  Letter := High(Letter);
  for I := 1 to 66 do
    Inc(Letter);
end;
```

`Letter` has the value A (ASCII 65).

Boolean Types

The four predefined Boolean types are **Boolean**, **ByteBool**, **WordBool**, and **LongBool**. **Boolean** is the preferred type. The others exist to provide compatibility with other languages and operating system libraries.

A **Boolean** variable occupies one byte of memory, a **ByteBool** variable also occupies one byte, a **WordBool** variable occupies two bytes (one word), and a **LongBool** variable occupies four bytes (two words).

Boolean values are denoted by the predefined constants **True** and **False**. The following relationships hold.

Boolean	**ByteBool, WordBool, LongBool**
False < True	*False <> True*
Ord(False) = 0	*Ord(False) = 0*
Ord(True) = 1	*Ord(True) <> 0*
Succ(False) = True	*Succ(False) = True*
Pred(True) = False	*Pred(False) = True*

A value of type **ByteBool**, **LongBool**, or **WordBool** is considered **True** when its ordinality is nonzero. If such a value appears in a context where a **Boolean** is expected, the compiler automatically converts any value of nonzero ordinality to True.

The previous remarks refer to the ordinality of Boolean values, not to the values themselves. In Delphi, Boolean expressions cannot be equated with integers or reals. Hence, if X is an integer variable, the statement

```
if X then ...;
```

generates a compilation error. Casting the variable to a Boolean type is unreliable, but each of the following alternatives will work.

```
if X <> 0 then ...; { use longer expression that returns Boolean value }
```

```
var OK: Boolean;
 ...
if X <> 0 then OK := True;
if OK then ...;
```

Enumerated Types

An enumerated type defines an ordered set of values by simply listing identifiers that denote these values. The values have no inherent meaning. To declare an enumerated type, use the syntax

type*typeName*`= (`*val1*`, ...,`*valn*`)`

where *typeName* and each *val* are valid identifiers. For example, the declaration

```
type Suit = (Club, Diamond, Heart, Spade);
```

defines an enumerated type called `Suit` whose possible values are `Club, Diamond`, `Heart`, and `Spade`, where `Ord(Club)` returns 0, `Ord(Diamond)` returns 1, and so forth.

When you declare an enumerated type, you are declaring each val to be a constant of type *typeName*. If the *val* identifiers are used for another purpose within the same scope, naming conflicts occur. For example, suppose you declare the type

```
type TSound = (Click, Clack, Clock)
```

Unfortunately, `Click` is also the name of a method defined for TControl and all of the objects in VCL that descend from it. So if you're writing an application and you create an event handler like

```
procedure TForm1.DBGridEnter(Sender: TObject);
var Thing: TSound;
begin
   ...
   Thing := Click;
end;
```

you'll get a compilation error; the compiler interprets `Click` within the scope of the procedure as a reference to `TForm`'s `Click` method. You can work around this by qualifying the identifier; thus, if `TSound` is declared in `MyUnit`, you would use

```
Thing := MyUnit.Click;
```

A better solution, however, is to choose constant names that are not likely to conflict with other identifiers. Examples:

```
type
  TSound = (tsClick, tsClack, tsClock);
  TMyColor = (mcRed, mcBlue, mcGreen, mcYellow, mcOrange);
  Answer = (ansYes, ansNo, ansMaybe)
```

You can use the *(val1, ..., valn)* construction directly in variable declarations, as if it were a type name:

```
var MyCard: (Club, Diamond, Heart, Spade);
```

But if you declare `MyCard` this way, you can't declare another variable within the same scope using these constant identifiers. Thus

```
var Card1: (Club, Diamond, Heart, Spade);
var Card2: (Club, Diamond, Heart, Spade);
```

generates a compilation error. But

```
var Card1, Card2: (Club, Diamond, Heart, Spade);
```

compiles cleanly, as does

```
type Suit = (Club, Diamond, Heart, Spade);
var
  Card1: Suit;
  Card2: Suit;
```

Enumerated Types with Explicitly Assigned Ordinality

By default, the ordinalities of enumerated values start from 0 and follow the sequence in which their identifiers are listed in the type declaration. You can override this by explicitly assigning ordinalities to some or all of the values in the declaration. To assign an ordinality to a value, follow its identifier with = *constantExpression*, where *constantExpression* is a constant expression (see page 589) that evaluates to an integer. For example,

```
type Size = (Small = 5, Medium = 10, Large = Small + Medium);
```

defines a type called `Size` whose possible values include `Small`, `Medium`, and `Large`, where `Ord(Small)` returns 5, `Ord(Medium)` returns 10, and `Ord(Large)` returns 15.

An enumerated type is, in effect, a subrange whose lowest and highest values correspond to the lowest and highest ordinalities of the constants in the declaration. In the previous example, the `Size` type has 11 possible values whose ordinalities range from 5 to 15. (Hence the type `array[Size] of Char` represents an array of 11 characters.) Only three of these values have names, but the others are accessible through typecasts and through routines such as `Pred`, `Succ`, `Inc`, and `Dec`. In the following example, "anonymous" values in the range of `Size` are assigned to the variable X.

```
var X: Size;

  X := Small;    // Ord(X) = 5
  Y := Size(6);  // Ord(X) = 6
  Inc(X);        // Ord(X) = 7
```

Any value that isn't explicitly assigned an ordinality has ordinality one greater than that of the previous value in the list. If the first value isn't assigned an ordinality, its ordinality is 0. Hence, given the declaration

```
type SomeEnum = (e1, e2, e3 = 1);
```

`SomeEnum` has only two possible values: `Ord(e1)` returns 0, `Ord(e2)` returns 1, and `Ord(e3)` also returns 1; because `e2` and `e3` have the same ordinality, they represent the same value.

Enumerated constants without a specific value have RTTI:

```
type SomeEnum = (e1, e2, e3);
```

whereas enumerated constants with a specific value, such as the following, do not have RTTI:

```
type SomeEnum = (e1 = 1, e2 = 2, e3 = 3);
```

Subrange Types

A subrange type represents a subset of the values in another ordinal type (called the base type). Any construction of the form *Low..High*, where *Low* and *High* are constant expressions of the same ordinal type and *Low* is less than *High*, identifies a subrange type that includes all values between *Low* and *High*. For example, if you declare the enumerated type

```
type TColors = (Red, Blue, Green, Yellow, Orange, Purple, White, Black);
```

you can then define a subrange type like

```
type TMyColors = Green..White;
```

Here `TMyColors` includes the values `Green`,`Yellow`, `Orange`, `Purple`, and `White`.

You can use numeric constants and characters (string constants of length 1) to define subrange types:

```
type
  SomeNumbers = -128..127;
  Caps = 'A'..'Z';
```

When you use numeric or character constants to define a subrange, the base type is the smallest integer or character type that contains the specified range.

The *LowerBound..UpperBound* construction itself functions as a type name, so you can use it directly in variable declarations. For example,

```
var SomeNum: 1..500;
```

declares an integer variable whose value can be anywhere in the range from 1 to 500.

The ordinality of each value in a subrange is preserved from the base type. (In the first example, if `Color` is a variable that holds the value `Green`, `Ord(Color)` returns 2 regardless of whether `Color` is of type `TColors` or `TMyColors`.) Values do not wrap around the beginning or end of a subrange, even if the base is an integer or character type; incrementing or decrementing past the boundary of a subrange simply converts the value to the base type. Hence, while

```
type  Percentile = 0..99;
var  I: Percentile;
  ...
  I := 100;
```

produces an error,

```
...
  I := 99;
  Inc(I);
```

assigns the value 100 to I (unless compiler range-checking is enabled).

The use of constant expressions in subrange definitions introduces a syntactic difficulty. In any type declaration, when the first meaningful character after **=** is a left parenthesis, the compiler assumes that an enumerated type is being defined. Hence the code

```
const
  X = 50;
  Y = 10;

type
  Scale = (X - Y) * 2..(X + Y) * 2;
```

produces an error. Work around this problem by rewriting the type declaration to avoid the leading parenthesis:

```
type
  Scale = 2 * (X - Y)..(X + Y) * 2;
```

Real Types

A real type defines a set of numbers that can be represented with floating-point notation. The table below gives the ranges and storage formats for the fundamental real types on the Win32 platform.

Fundamental Win32 real types

Type	Range	Significant digits	Size in bytes
Real48	-2.9 x 10^–39 .. 1.7 x 10^38	11-12	6
Single	-1.5 x 10^–45 .. 3.4 x 10^38	7-8	4
Double	-5.0 x 10^–324 .. 1.7 x 10^308	15-16	8
Extended	-3.6 x 10^–4951 .. 1.1 x 10^4932	10-20	10
Comp	-2^63+1 .. 2^63–1	10-20	8
Currency	-922337203685477.5808.. 922337203685477.5807	10-20	8

The following table shows how the fundamental real types map to .NET framework types.

Fundamental .NET real type mappings

Type	.NET Mapping
Real48	Deprecated
Single	Single
Double	Double
Extended	Double
Comp	Deprecated
Currency	Re-implemented as a value type using the Decimal type from the .NET Framework

The generic type **Real**, in its current implementation, is equivalent to **Double** (which maps to Double on .NET).

Generic real types

Type	Range	Significant digits	Size in bytes
Real	-5.0 x 10^–324 .. 1.7 x 10^308	15–16	8

Note: The six-byte Real48

type was called **Real** in earlier versions of Object Pascal. If you are recompiling code that uses the older, six-byte **Real** type in Delphi, you may want to change it to **Real48**. You can also use the `{$REALCOMPATIBILITY ON}` compiler directive to turn **Real** back into the six-byte type. The following remarks apply to fundamental real types.

- **Real48** is maintained for backward compatibility. Since its storage format is not native to the Intel processor architecture, it results in slower performance than other floating-point types. The **Real48** type has been deprecated on the .NET platform.
- **Extended** offers greater precision than other real types but is less portable. Be careful using **Extended** if you are creating data files to share across platforms.
- The **Comp** (computational) type is native to the Intel processor architecture and represents a 64-bit integer. It is classified as a real, however, because it does not behave like an ordinal type. (For example, you cannot increment or decrement a **Comp** value.) **Comp** is maintained for backward compatibility only. Use the **Int64** type for better performance.
- **Currency** is a fixed-point data type that minimizes rounding errors in monetary calculations. On the Win32 platform, it is stored as a scaled 64-bit integer with the four least significant digits implicitly representing decimal places. When mixed with other real types in assignments and expressions, **Currency** values are automatically divided or multiplied by 10000.

See Also

Data Types (↗ see page 553)

String Types (↗ see page 561)

Structured Types (↗ see page 566)

Pointers and Pointer Types (↗ see page 575)

Procedural Types (↗ see page 578)

Variant Types (↗ see page 580)

Type Compatibility and Identity (↗ see page 583)

Declaring Types (↗ see page 586)

Variables (↗ see page 587)

Declared Constants (↗ see page 589)

3.1.3.2.3 String Types

This topic describes the string data types available in the Delphi language. The following types are covered:

- Short strings.
- Long strings.
- Wide (Unicode) strings.

About String Types

A string represents a sequence of characters. Delphi supports the following predefined string types.

String types

Type	Maximum length	Memory required	Used for
ShortString	255 characters	2 to 256 bytes	backward compatibility
AnsiString	~2^31 characters	4 bytes to 2GB	8-bit (ANSI) characters, DBCS ANSI, MBCS ANSI, etc.
WideString	~2^30 characters	4 bytes to 2GB	Unicode characters; multi-user servers and multi-language applications

On the Win32 platform, **AnsiString**, sometimes called the long string, is the preferred type for most purposes. **WideString** is the preferred string type on the .NET platform.

String types can be mixed in assignments and expressions; the compiler automatically performs required conversions. But strings passed by reference to a function or procedure (as **var** and **out** parameters) must be of the appropriate type. Strings can be explicitly cast to a different string type.

The reserved word **string** functions like a generic type identifier. For example,

```
var S: string;
```

creates a variable S that holds a string. On the Win32 platform, the compiler interprets **string** (when it appears without a bracketed number after it) as **AnsiString**. On the .NET platform, the **string** type maps to the String class. You can use single byte character strings on the .NET platform, but you must explicitly declare them to be of type **AnsiString**.

On the Win32 platform, you can use the {$H-} directive to turn **string** into **ShortString**. The {$H-} directive is deprecated on the .NET platform.

The standard function Length returns the number of characters in a string. The SetLength procedure adjusts the length of a string.

Comparison of strings is defined by the ordering of the characters in corresponding positions. Between strings of unequal length, each character in the longer string without a corresponding character in the shorter string takes on a greater-than value. For example, 'AB' is greater than 'A'; that is, 'AB' > 'A' returns **True**. Zero-length strings hold the lowest values.

You can index a string variable just as you would an array. If S is a string variable and i an integer expression, S[i] represents the *ith* character - or, strictly speaking, the *ith* byte in S. For a **ShortString** or **AnsiString**, S[i] is of type **AnsiChar**; for a **WideString**, S[i] is of type **WideChar**. For single-byte (Western) locales, MyString[2] := 'A'; assigns the value A to the second character of MyString. The following code uses the standard AnsiUpperCase function to convert MyString to uppercase.

```
var I: Integer;
begin
   I := Length(MyString);
   while I > 0 do
    begin
       MyString[I] := AnsiUpperCase(MyString[I]);
       I := I - 1;
    end;
end;
```

Be careful indexing strings in this way, since overwriting the end of a string can cause access violations. Also, avoid passing long-string indexes as **var** parameters, because this results in inefficient code.

You can assign the value of a string constant - or any other expression that returns a string - to a variable. The length of the string changes dynamically when the assignment is made. Examples:

```
MyString := 'Hello world!';
MyString := 'Hello' + 'world';
MyString := MyString + '!';
MyString := ' '; { space }
MyString := '';  { empty string }
```

Short Strings

A **ShortString** is 0 to 255 characters long. While the length of a **ShortString** can change dynamically, its memory is a statically allocated 256 bytes; the first byte stores the length of the string, and the remaining 255 bytes are available for characters. If `S` is a **ShortString** variable, `Ord(S[0])`, like `Length(S)`, returns the length of `S`; assigning a value to `S[0]`, like calling `SetLength`, changes the length of `S`. **ShortString** is maintained for backward compatibility only.

The Delphi language supports short-string types - in effect, subtypes of **ShortString** - whose maximum length is anywhere from 0 to 255 characters. These are denoted by a bracketed numeral appended to the reserved word **string**. For example,

```
var MyString: string[100];
```

creates a variable called `MyString` whose maximum length is 100 characters. This is equivalent to the declarations

```
type CString = string[100];
var MyString: CString;
```

Variables declared in this way allocate only as much memory as the type requires - that is, the specified maximum length plus one byte. In our example, `MyString` uses 101 bytes, as compared to 256 bytes for a variable of the predefined **ShortString** type.

When you assign a value to a short-string variable, the string is truncated if it exceeds the maximum length for the type.

The standard functions **High** and **Low** operate on short-string type identifiers and variables. **High** returns the maximum length of the short-string type, while **Low** returns zero.

Long Strings

AnsiString, also called a long string, represents a dynamically allocated string whose maximum length is limited only by available memory.

A long-string variable is a pointer occupying four bytes of memory. When the variable is empty - that is, when it contains a zero-length stringthe pointer is **nil** and the string uses no additional storage. When the variable is nonempty, it points a dynamically allocated block of memory that contains the string value. The eight bytes before the location contain a 32-bit length indicator and a 32-bit reference count. This memory is allocated on the heap, but its management is entirely automatic and requires no user code.

Because long-string variables are pointers, two or more of them can reference the same value without consuming additional memory. The compiler exploits this to conserve resources and execute assignments faster. Whenever a long-string variable is destroyed or assigned a new value, the reference count of the old string (the variable's previous value) is decremented and the reference count of the new value (if there is one) is incremented; if the reference count of a string reaches zero, its memory is deallocated. This process is called reference-counting. When indexing is used to change the value of a single character in a string, a copy of the string is made if - but only if - its reference count is greater than one. This is called copy-on-write semantics.

WideString

The **WideString** type represents a dynamically allocated string of 16-bit Unicode characters. In most respects it is similar to **AnsiString**. On Win32, **WideString** is compatible with the COM BSTR type.

Note: Under Win32, WideString values are not reference-counted.

The Win32 platform supports single-byte and multibyte character sets as well as Unicode. With a single-byte character set (SBCS), each byte in a string represents one character.

In a multibyte character set (MBCS), some characters are represented by one byte and others by more than one byte. The first byte of a multibyte character is called the lead byte. In general, the lower 128 characters of a multibyte character set map to the 7-bit ASCII characters, and any byte whose ordinal value is greater than 127 is the lead byte of a multibyte character. The null value (#0) is always a single-byte character. Multibyte character sets - especially double-byte character sets (DBCS) - are widely used for Asian languages.

In the Unicode character set, each character is represented by two bytes. Thus a Unicode string is a sequence not of individual bytes but of two-byte words. Unicode characters and strings are also called wide characters and wide character strings. The first 256 Unicode characters map to the ANSI character set. The Windows operating system supports Unicode (UCS-2).

The Delphi language supports single-byte and multibyte characters and strings through the **Char**, **PChar**, **AnsiChar**, **PAnsiChar**, and **AnsiString** types. Indexing of multibyte strings is not reliable, since `S[i]` represents the *ith* byte (not necessarily the *ith* character) in S. However, the standard string-handling functions have multibyte-enabled counterparts that also implement locale-specific ordering for characters. (Names of multibyte functions usually start with `Ansi-`. For example, the multibyte version of `StrPos` is `AnsiStrPos`.) Multibyte character support is operating-system dependent and based on the current locale.

Delphi supports Unicode characters and strings through the **WideChar**, **PWideChar**, and **WideString** types.

Working with null-Terminated Strings

Many programming languages, including C and C++, lack a dedicated string data type. These languages, and environments that are built with them, rely on null-terminated strings. A null-terminated string is a zero-based array of characters that ends with NUL (#0); since the array has no length indicator, the first NUL character marks the end of the string. You can use Delphi constructions and special routines in the `SysUtils` unit (see Standard routines and I/O (see page 692)) to handle null-terminated strings when you need to share data with systems that use them.

For example, the following type declarations could be used to store null-terminated strings.

```
type
  TIdentifier = array[0..15] of Char;
  TFileName = array[0..259] of Char;
  TMemoText = array[0..1023] of WideChar;
```

With extended syntax enabled (`{$X+}`), you can assign a string constant to a statically allocated zero-based character array. (Dynamic arrays won't work for this purpose.) If you initialize an array constant with a string that is shorter than the declared length of the array, the remaining characters are set to `#0`.

Using Pointers, Arrays, and String Constants

To manipulate null-terminated strings, it is often necessary to use pointers. (See Pointers and pointer types (see page 575).) String constants are assignment-compatible with the **PChar** and **PWideChar** types, which represent pointers to null-terminated arrays of **Char** and **WideChar** values. For example,

```
var P: PChar;
  ...
P := 'Hello world!'
```

points P to an area of memory that contains a null-terminated copy of 'Hello world!' This is equivalent to

```
const TempString: array[0..12] of Char = 'Hello world!';
var P: PChar;
   ...
P := @TempString[0];
```

You can also pass string constants to any function that takes value or **const** parameters of type **PChar** or **PWideChar** - for example `StrUpper('Hello world!')`. As with assignments to a **PChar**, the compiler generates a null-terminated copy of the string and gives the function a pointer to that copy. Finally, you can initialize **PChar** or **PWideChar** constants with string literals, alone or in a structured type. Examples:

```
const
  Message: PChar = 'Program terminated';
  Prompt: PChar = 'Enter values: ';
  Digits: array[0..9] of PChar = ('Zero', 'One', 'Two', 'Three', 'Four', 'Five', 'Six',
'Seven', 'Eight', 'Nine');
```

Zero-based character arrays are compatible with **PChar** and **PWideChar**. When you use a character array in place of a pointer value, the compiler converts the array to a pointer constant whose value corresponds to the address of the first element of the array. For example,

```
var
  MyArray: array[0..32] of Char;
  MyPointer: PChar;
begin
  MyArray := 'Hello';
  MyPointer := MyArray;
  SomeProcedure(MyArray);
  SomeProcedure(MyPointer);
end;
```

This code calls `SomeProcedure` twice with the same value.

A character pointer can be indexed as if it were an array. In the previous example, `MyPointer[0]` returns `H`. The index specifies an offset added to the pointer before it is dereferenced. (For **PWideChar** variables, the index is automatically multiplied by two.) Thus, if `P` is a character pointer, `P[0]` is equivalent to `P^` and specifies the first character in the array, `P[1]` specifies the second character in the array, and so forth; `P[-1]` specifies the 'character' immediately to the left of `P[0]`. The compiler performs no range checking on these indexes.

The `StrUpper` function illustrates the use of pointer indexing to iterate through a null-terminated string:

```
function StrUpper(Dest, Source: PChar; MaxLen: Integer): PChar;
var
  I: Integer;
begin
  I := 0;
  while (I < MaxLen) and (Source[I] <> #0) do
  begin
    Dest[I] := UpCase(Source[I]);
    Inc(I);
  end;
  Dest[I] := #0;
  Result := Dest;
end;
```

Mixing Delphi Strings and Null-Terminated Strings

You can mix long strings (**AnsiString** values) and null-terminated strings (**PChar** values) in expressions and assignments, and you can pass **PChar** values to functions or procedures that take long-string parameters. The assignment `S := P`, where `S` is a string variable and `P` is a **PChar** expression, copies a null-terminated string into a long string.

In a binary operation, if one operand is a long string and the other a **PChar**, the **PChar** operand is converted to a long string.

You can cast a **PChar** value as a long string. This is useful when you want to perform a string operation on two **PChar** values. For example,

```
S := string(P1) + string(P2);
```

You can also cast a long string as a null-terminated string. The following rules apply.

- If `S` is a long-string expression, `PChar(S)` casts `S` as a null-terminated string; it returns a pointer to the first character in `S`. For example, if `Str1` and `Str2` are long strings, you could call the Win32 API `MessageBox` function like this: `MessageBox(0, PChar(Str1), PChar(Str2), MB_OK);`
- You can also use `Pointer(S)` to cast a long string to an untyped pointer. But if `S` is empty, the typecast returns **nil**.
- `PChar(S)` always returns a pointer to a memory block; if `S` is empty, a pointer to `#0` is returned.
- When you cast a long-string variable to a pointer, the pointer remains valid until the variable is assigned a new value or goes out of scope. If you cast any other long-string expression to a pointer, the pointer is valid only within the statement where the typecast is performed.
- When you cast a long-string expression to a pointer, the pointer should usually be considered read-only. You can safely use

the pointer to modify the long string only when all of the following conditions are satisfied.

- The expression cast is a long-string variable.
- The string is not empty.
- The string is unique - that is, has a reference count of one. To guarantee that the string is unique, call the **SetLength**, `SetString`, or `UniqueString` procedure.
- The string has not been modified since the typecast was made.
- The characters modified are all within the string. Be careful not to use an out-of-range index on the pointer.

The same rules apply when mixing **WideString** values with **PWideChar** values.

See Also

Data Types (see page 553)

Simple Types (see page 554)

Structured Types (see page 566)

Pointers and Pointer Types (see page 575)

Procedural Types (see page 578)

Variant Types (see page 580)

Type Compatibility and Identity (see page 583)

Declaring Types (see page 586)

Variables (see page 587)

Declared Constants (see page 589)

3.1.3.2.4 Structured Types

Instances of a structured type hold more than one value. Structured types include sets, arrays, records, and files as well as class, class-reference, and interface types. Except for sets, which hold ordinal values only, structured types can contain other structured types; a type can have unlimited levels of structuring.

Note: Typed and untyped file types are not supported with the .NET framework.

By default, the values in a structured type are aligned on word or double-word boundaries for faster access. When you declare a structured type, you can include the reserved word **packed** to implement compressed data storage. For example, `type TNumbers = packed array [1..100] of Real;`

Using **packed** slows data access and, in the case of a character array, affects type compatibility (for more information, see Memory management (see page 644)).

This topic covers the following structured types:

- Sets
- Arrays, including static and dynamic arrays.
- Records
- File types

Sets

A set is a collection of values of the same ordinal type. The values have no inherent order, nor is it meaningful for a value to be included twice in a set.

The range of a set type is the power set of a specific ordinal type, called the base type; that is, the possible values of the set type are all the subsets of the base type, including the empty set. The base type can have no more than 256 possible values, and their ordinalities must fall between 0 and 255. Any construction of the form

`set of`*baseType*

where *baseType* is an appropriate ordinal type, identifies a set type.

Because of the size limitations for base types, set types are usually defined with subranges. For example, the declarations

```
type
 TSomeInts = 1..250;
 TIntSet = set of TSomeInts;
```

create a set type called `TIntSet` whose values are collections of integers in the range from 1 to 250. You could accomplish the same thing with

```
type TIntSet = set of 1..250;
```

Given this declaration, you can create a sets like this:

```
var Set1, Set2: TIntSet;
    ...
    Set1 := [1, 3, 5, 7, 9];
    Set2 := [2, 4, 6, 8, 10]
```

You can also use the `set of ...` construction directly in variable declarations:

```
var MySet: set of 'a'..'z';
    ...
    MySet := ['a','b','c'];
```

Other examples of set types include

```
set of Byte
set of (Club, Diamond, Heart, Spade)
set of Char;
```

The **in** operator tests set membership:

```
if 'a' in MySet then ... { do something } ;
```

Every set type can hold the empty set, denoted by [].

Arrays

An array represents an indexed collection of elements of the same type (called the base type). Because each element has a unique index, arrays, unlike sets, can meaningfully contain the same value more than once. Arrays can be allocated *statically* or *dynamically*.

Static Arrays

Static array types are denoted by constructions of the form

`array[`*indexType1*, ..., *indexTypen*`] of` *baseType*;

where each *indexType* is an ordinal type whose range does not exceed 2GB. Since the *indexTypes* index the array, the number of elements an array can hold is limited by the product of the sizes of the *indexTypes*. In practice, *indexTypes* are usually integer subranges.

In the simplest case of a one-dimensional array, there is only a single *indexType*. For example,

```
var MyArray: array [1..100] of Char;
```

declares a variable called `MyArray` that holds an array of 100 character values. Given this declaration, `MyArray[3]` denotes the third character in `MyArray`. If you create a static array but don't assign values to all its elements, the unused elements are still allocated and contain random data; they are like uninitialized variables.

A multidimensional array is an array of arrays. For example,

```
type TMatrix = array[1..10] of array[1..50] of Real;
```

is equivalent to

```
type TMatrix = array[1..10, 1..50] of Real;
```

Whichever way `TMatrix` is declared, it represents an array of 500 real values. A variable `MyMatrix` of type `TMatrix` can be indexed like this: `MyMatrix[2,45];` or like this: `MyMatrix[2][45]`. Similarly,

```
packed array[Boolean, 1..10, TShoeSize] of Integer;
```

is equivalent to

```
packed array[Boolean] of packed array[1..10] of packed array[TShoeSize] of Integer;
```

The standard functions `Low` and `High` operate on array type identifiers and variables. They return the low and high bounds of the array's first index type. The standard function `Length` returns the number of elements in the array's first dimension.

A one-dimensional, packed, static array of **Char** values is called a packed string. Packed-string types are compatible with string types and with other packed-string types that have the same number of elements. See Type compatibility and identity (↗ see page 583).

An array type of the form `array[0..x] of Char` is called a zero-based character array. Zero-based character arrays are used to store null-terminated strings and are compatible with **PChar** values. See Working with null-terminated strings (↗ see page 561).

Dynamic Arrays

Dynamic arrays do not have a fixed size or length. Instead, memory for a dynamic array is reallocated when you assign a value to the array or pass it to the `SetLength` procedure. Dynamic-array types are denoted by constructions of the form

`array of` *baseType*

For example,

```
var MyFlexibleArray: array of Real;
```

declares a one-dimensional dynamic array of reals. The declaration does not allocate memory for `MyFlexibleArray`. To create the array in memory, call `SetLength`. For example, given the previous declaration,

```
SetLength(MyFlexibleArray, 20);
```

allocates an array of 20 reals, indexed 0 to 19. Dynamic arrays are always integer-indexed, always starting from 0.

Dynamic-array variables are implicitly pointers and are managed by the same reference-counting technique used for long strings. To deallocate a dynamic array, assign **nil** to a variable that references the array or pass the variable to `Finalize`; either of these methods disposes of the array, provided there are no other references to it. Dynamic arrays are automatically released when their reference-count drops to zero. Dynamic arrays of length 0 have the value **nil**. Do not apply the dereference operator (**^**) to a dynamic-array variable or pass it to the `New` or `Dispose` procedure.

If `X` and `Y` are variables of the same dynamic-array type, `X := Y` points `X` to the same array as `Y`. (There is no need to allocate memory for `X` before performing this operation.) Unlike strings and static arrays, *copy-on-write* is not employed for dynamic arrays, so they are not automatically copied before they are written to. For example, after this code executes,

```
var
  A, B: array of Integer;
  begin
    SetLength(A, 1);
    A[0] := 1;
    B := A;
    B[0] := 2;
  end;
```

the value of `A[0]` is 2. (If `A` and `B` were static arrays, `A[0]` would still be 1.)

Assigning to a dynamic-array index (for example, `MyFlexibleArray[2] := 7`) does not reallocate the array. Out-of-range indexes are not reported at compile time.

In contrast, to make an independent copy of a dynamic array, you must use the global `Copy` function:

```
var
  A, B: array of Integer;
  begin
    SetLength(A, 1);
    A[0] := 1;
    B := Copy(A);
    B[0] := 2; { B[0] <> A[0] }
  end;
```

When dynamic-array variables are compared, their references are compared, not their array values. Thus, after execution of the code

```
var
  A, B: array of Integer;
  begin
     SetLength(A, 1);
     SetLength(B, 1);
     A[0] := 2;
     B[0] := 2;
  end;
```

`A = B` returns **False** but `A[0] = B[0]` returns **True**.

To truncate a dynamic array, pass it to `SetLength`, or pass it to `Copy` and assign the result back to the array variable. (The `SetLength` procedure is usually faster.) For example, if A is a dynamic array, `A := SetLength(A, 0, 20)` truncates all but the first 20 elements of A.

Once a dynamic array has been allocated, you can pass it to the standard functions `Length`, `High`, and `Low`. `Length` returns the number of elements in the array, `High` returns the array's highest index (that is, `Length - 1`), and `Low` returns 0. In the case of a zero-length array, `High` returns 1 (with the anomalous consequence that `High < Low`).

Note: In some function and procedure declarations, array parameters are represented as `array of` *baseType*, without any index types specified. For example,`function CheckStrings(A: array of string): Boolean;`

This indicates that the function operates on all arrays of the specified base type, regardless of their size, how they are indexed, or whether they are allocated statically or dynamically.

Multidimensional Dynamic Arrays

To declare multidimensional dynamic arrays, use iterated `array of ...` constructions. For example,

```
type TMessageGrid = array of array of string;
var Msgs: TMessageGrid;
```

declares a two-dimensional array of strings. To instantiate this array, call `SetLength` with two integer arguments. For example, if `I` and `J` are integer-valued variables,

```
SetLength(Msgs,I,J);
```

allocates an *I-by-J* array, and `Msgs[0,0]` denotes an element of that array.

You can create multidimensional dynamic arrays that are not rectangular. The first step is to call `SetLength`, passing it parameters for the first n dimensions of the array. For example,

```
var Ints: array of array of Integer;
SetLength(Ints,10);
```

allocates ten rows for `Ints` but no columns. Later, you can allocate the columns one at a time (giving them different lengths); for example

```
SetLength(Ints[2], 5);
```

makes the third column of `Ints` five integers long. At this point (even if the other columns haven't been allocated) you can assign values to the third column - for example, `Ints[2,4] := 6`.

The following example uses dynamic arrays (and the IntToStr function declared in the `SysUtils` unit) to create a triangular matrix of strings.

```
var
  A : array of array of string;
  I, J : Integer;
  begin
    SetLength(A, 10);
    for I := Low(A) to High(A) do
      begin
        SetLength(A[I], I);
        for J := Low(A[I]) to High(A[I]) do
          A[I,J] := IntToStr(I) + ',' + IntToStr(J) + ' ';
      end;
  end;
```

Array Types and Assignments

Arrays are assignment-compatible only if they are of the same type. Because the Delphi language uses name-equivalence for types, the following code will not compile.

```
var
  Int1: array[1..10] of Integer;
  Int2: array[1..10] of Integer;
      ...
  Int1 := Int2;
```

To make the assignment work, declare the variables as

```
var Int1, Int2: array[1..10] of Integer;
```

or

```
type IntArray = array[1..10] of Integer;
var
   Int1: IntArray;
   Int2: IntArray;
```

Dynamically Allocated Multidimensional Arrays (.NET)

On the .NET platform, multidimensional arrays can be dynamically allocated using the New standard function. Using the New syntax to allocate an array, the array declaration specifies the number of dimensions, but not their actual size. You then pass the element type, the actual array dimensions, or an array initializer list to the New function. The array declaration has the following syntax:

`array[, ..., ] of` *baseType*;

In the syntax, the number of dimensions are specified by using a comma as a placeholder. The actual size is not determined until runtime when you call the New function. There are two forms of the New function: One takes the element type and the size of the array, and the other takes the element type and an array initializer list. The following code demonstrates both forms:

```
var
  a: array [,,] of integer;  // 3 dimensional array
  b: array [,] of integer;   // 2 dimensional array
  c: array [,] of TPoint;    // 2 dimensional array of TPoint

begin
   a := New(array[3,5,7] of integer);                 // New taking element type and size of
each dimension.
```

```
   b := New(array[,] of integer, ((1,2,3), (4,5,6))); // New taking the element type and
initializer list.
                                                       // New taking an initializer list of
TPoint.
   c := New(array[,] of TPoint, (((X:1;Y:2), (X:3;Y:4)), ((X:5;Y:6), (X:7;Y:8))));
end.
```

You can allocate the array by passing variable or constant expressions to the New function:

```
var
   a:   array[,] of integer;
   r,c: Integer;

begin
   r := 4;
   c := 17;

   a := New(array [r,c] of integer);
```

You can also use the SetLength procedure to allocate the array by passing the array expression and the size of each dimension:

```
var
  a: array[,] of integer;
  b: array[,,] of integer;

begin
     SetLength(a, 4,5);
     SetLength(b, 3,5,7);
end.
```

The Copy function can be used to make a copy of an entire array. You cannot use Copy to duplicate a portion of an array.

You cannot pass a dynamically allocated rectangular array to the Low or High functions. This will generate a compile-time error.

Records (traditional)

A **record** (analogous to a structure in some languages) represents a heterogeneous set of elements. Each element is called a field; the declaration of a record type specifies a name and type for each field. The syntax of a record type declaration is

```
type recordTypeName = record
       fieldList1: type1;
        ...
       fieldListn: typen;
     end
```

where *recordTypeName* is a valid identifier, each type denotes a type, and each fieldList is a valid identifier or a comma-delimited list of identifiers. The final semicolon is optional.

For example, the following declaration creates a record type called `TDateRec`.

```
type
  TDateRec = record
    Year: Integer;
    Month: (Jan, Feb, Mar, Apr, May, Jun,
            Jul, Aug, Sep, Oct, Nov, Dec);
    Day: 1..31;
  end;
```

Each `TDateRec` contains three fields: an integer value called `Year`, a value of an enumerated type called `Month`, and another integer between `1` and `31` called `Day`. The identifiers `Year`, `Month`, and `Day` are the field designators for `TDateRec`, and they behave like variables. The `TDateRec` type declaration, however, does not allocate any memory for the `Year`, `Month`, and `Day` fields; memory is allocated when you instantiate the record, like this:

```
var Record1, Record2: TDateRec;
```

This variable declaration creates two instances of `TDateRec`, called `Record1` and `Record2`.

You can access the fields of a record by qualifying the field designators with the record's name:

```
Record1.Year := 1904;
Record1.Month := Jun;
Record1.Day := 16;
```

Or use a **with** statement:

```
with Record1 do
 begin
    Year := 1904;
    Month := Jun;
    Day := 16;
 end;
```

You can now copy the values of `Record1`'s fields to `Record2`:

```
Record2 := Record1;
```

Because the scope of a field designator is limited to the record in which it occurs, you don't have to worry about naming conflicts between field designators and other variables.

Instead of defining record types, you can use the `record ...` construction directly in variable declarations:

```
var S: record
      Name: string;
      Age: Integer;
    end;
```

However, a declaration like this largely defeats the purpose of records, which is to avoid repetitive coding of similar groups of variables. Moreover, separately declared records of this kind will not be assignment-compatible, even if their structures are identical.

Variant Parts in Records

A record type can have a variant part, which looks like a **case** statement. The variant part must follow the other fields in the record declaration.

To declare a record type with a variant part, use the following syntax.

```
type recordTypeName = record
       fieldList1: type1;
        ...
       fieldListn: typen;
       case tag: ordinalType of
          constantList1: (variant1);
          ...
          constantListn: (variantn);
     end;
```

The first part of the declaration - up to the reserved word **case** - is the same as that of a standard record type. The remainder of the declaration - from **case** to the optional final semicolon - is called the variant part. In the variant part,

- *tag* is optional and can be any valid identifier. If you omit *tag*, omit the colon (**:**) after it as well.
- *ordinalType* denotes an ordinal type.
- Each *constantList* is a constant denoting a value of type *ordinalType*, or a comma-delimited list of such constants. No value can be represented more than once in the combined *constantLists*.
- Each *variant* is a semicolon-delimited list of declarations resembling the *fieldList: type* constructions in the main part of the record type. That is, a variant has the form

```
fieldList1: type1;
 ...
fieldListn: typen;
```

where each *fieldList* is a valid identifier or comma-delimited list of identifiers, each type denotes a type, and the final semicolon is optional. The types must not be long strings, dynamic arrays, variants (that is, Variant types), or interfaces, nor can they be structured types that contain long strings, dynamic arrays, variants, or interfaces; but they can be pointers to these types.

Records with variant parts are complicated syntactically but deceptively simple semantically. The variant part of a record contains several variants which share the same space in memory. You can read or write to any field of any variant at any time; but if you write to a field in one variant and then to a field in another variant, you may be overwriting your own data. The tag, if there is one, functions as an extra field (of type *ordinalType*) in the non-variant part of the record.

Variant parts have two purposes. First, suppose you want to create a record type that has fields for different kinds of data, but you know that you will never need to use all of the fields in a single record instance. For example,

```
type
   TEmployee = record
      FirstName, LastName: string[40];
      BirthDate: TDate;
      case Salaried: Boolean of
        True: (AnnualSalary: Currency);
        False: (HourlyWage: Currency);
   end;
```

The idea here is that every employee has either a salary or an hourly wage, but not both. So when you create an instance of `TEmployee`, there is no reason to allocate enough memory for both fields. In this case, the only difference between the variants is in the field names, but the fields could just as easily have been of different types. Consider some more complicated examples:

```
type
  TPerson = record
    FirstName, LastName: string[40];
    BirthDate: TDate;
    case Citizen: Boolean of
        True: (Birthplace: string[40]);
        False: (Country: string[20];
                EntryPort: string[20];
                EntryDate, ExitDate: TDate);
  end;

type
  TShapeList = (Rectangle, Triangle, Circle, Ellipse, Other);
  TFigure = record
    case TShapeList of
         Rectangle: (Height, Width: Real);
         Triangle: (Side1, Side2, Angle: Real);
         Circle: (Radius: Real);
         Ellipse, Other: ();
  end;
```

For each record instance, the compiler allocates enough memory to hold all the fields in the largest variant. The optional tag and the *constantLists* (like `Rectangle`, `Triangle`, and so forth in the last example) play no role in the way the compiler manages the fields; they are there only for the convenience of the programmer.

The second reason for variant parts is that they let you treat the same data as belonging to different types, even in cases where the compiler would not allow a typecast. For example, if you have a 64-bit **Real** as the first field in one variant and a 32-bit **Integer** as the first field in another, you can assign a value to the **Real** field and then read back the first 32 bits of it as the value of the **Integer** field (passing it, say, to a function that requires integer parameters).

Records (advanced)

In addition to the traditional record types, the Delphi language allows more complex and “class-like” record types. In addition to fields, records may have properties and methods (including constructors), class properties, class methods, class fields, and nested types. For more information on these subjects, see the documentation on Classes and Objects. Here is a sample record type definition with some “class-like” functionality.

```
type
  TMyRecord = record
    type
      TInnerColorType = Integer;
    var
```

3

```
      Red: Integer;
    class var
      Blue: Integer;
    procedure printRed();
    constructor Create(val: Integer);
    property RedProperty: TInnerColorType read Red write Red;
    class property BlueProp: TInnerColorType read Blue write Blue;
end;

constructor TMyRecord.Create(val: Integer);
begin
  Red := val;
end;

procedure TMyRecord.printRed;
begin
  writeln('Red: ', Red);
end;
```

Though records can now share much of the functionality of classes, there are some important differences between classes and records.

- Records do not support inheritance.
- Records can contain variant parts; classes cannot.
- Records are value types, so they are copied on assignment, passed by value, and allocated on the stack unless they are declared globally or explicitly allocated using the `New` and `Dispose` function. Classes are reference types, so they are not copied on assignment, they are passed by reference, and they are allocated on the heap.
- Records allow operator overloading on the Win32 and .NET platforms; classes allow operator overloading only for .NET.
- Records are constructed automatically, using a default no-argument constructor, but classes must be explicitly constructed. Because records have a default no-argument constructor, any user-defined record constructor must have one or more parameters.
- Record types cannot have destructors.
- Virtual methods (those specified with the **virtual**, **dynamic**, and **message** keywords) cannot be used in record types.
- Unlike classes, record types on the Win32 platform cannot implement interfaces; however, records on the .NET platform can implement interfaces.

File Types (Win32)

File types, which are available only on the Win32 platform, are sequences of elements of the same type. Standard I/O routines use the predefined `TextFile` or `Text` type, which represents a file containing characters organized into lines. For more information about file input and output, see Standard routines and I/O (see page 692).

To declare a file type, use the syntax

type *fileTypeName* = **file of** *type*

where *fileTypeName* is any valid identifier and type is a fixed-size type. Pointer types - whether implicit or explicit - are not allowed, so a file cannot contain dynamic arrays, long strings, classes, objects, pointers, variants, other files, or structured types that contain any of these.

For example,

```
type
   PhoneEntry = record
     FirstName, LastName: string[20];
     PhoneNumber: string[15];
     Listed: Boolean;
   end;
   PhoneList = file of PhoneEntry;
```

declares a file type for recording names and telephone numbers.

You can also use the `file of ...` construction directly in a variable declaration. For example,

```
var List1: file of PhoneEntry;
```

The word **file** by itself indicates an untyped file:

```
var DataFile: file;
```

For more information, see Untyped files (see page 692).

Files are not allowed in arrays or records.

See Also

Data Types (see page 553)

Simple Types (see page 554)

String Types (see page 561)

Pointers and Pointer Types (see page 575)

Procedural Types (see page 578)

Variant Types (see page 580)

Type Compatibility and Identity (see page 583)

Declaring Types (see page 586)

Variables (see page 587)

Declared Constants (see page 589)

Classes and Objects (see page 513)

3.1.3.2.5 Pointers and Pointer Types

A pointer is a variable that denotes a memory address. When a pointer holds the address of another variable, we say that it points to the location of that variable in memory or to the data stored there. In the case of an array or other structured type, a pointer holds the address of the first element in the structure. If that address is already taken, then the pointer holds the address to the first element.

Pointers are typed to indicate the kind of data stored at the addresses they hold. The general-purpose **Pointer** type can represent a pointer to any data, while more specialized pointer types reference only specific types of data. Pointers occupy four bytes of memory.

This topic contains information on the following:

- General overview of pointer types.
- Declaring and using the pointer types supported by Delphi.

Overview of pointers

To see how pointers work, look at the following example.

```
1         var
2           X, Y: Integer;  // X and Y are Integer variables
3           P: ^Integer     // P points to an Integer
4           begin
5             X := 17;      // assign a value to X
6             P := @X;      // assign the address of X to P
7             Y := P^;      // dereference P; assign the result to Y
8           end;
```

Line 2 declares X and Y as variables of type **Integer**. Line 3 declares P as a pointer to an **Integer** value; this means that P can point to the location of `X` or `Y`. Line 5 assigns a value to `X`, and line 6 assigns the address of `X` (denoted by `@X`) to `P`. Finally, line 7 retrieves the value at the location pointed to by `P` (denoted by `^P`) and assigns it to `Y`. After this code executes, `X` and `Y` have the same value, namely 17.

The @ operator, which we have used here to take the address of a variable, also operates on functions and procedures. For more information, see The @ operator (see page 720) and Procedural types in statements and expressions (see page 578).

The symbol **^** has two purposes, both of which are illustrated in our example. When it appears before a type identifier

^typeName

it denotes a type that represents pointers to variables of type *typeName*. When it appears after a pointer variable

*pointer^*

it dereferences the pointer; that is, it returns the value stored at the memory address held by the pointer.

Our example may seem like a roundabout way of copying the value of one variable to another - something that we could have accomplished with a simple assignment statement. But pointers are useful for several reasons. First, understanding pointers will help you to understand the Delphi language, since pointers often operate behind the scenes in code where they don't appear explicitly. Any data type that requires large, dynamically allocated blocks of memory uses pointers. Long-string variables, for instance, are implicitly pointers, as are class instance variables. Moreover, some advanced programming techniques require the use of pointers.

Finally, pointers are sometimes the only way to circumvent Delphi's strict data typing. By referencing a variable with an all-purpose **Pointer**, casting the **Pointer** to a more specific type, and then dereferencing it, you can treat the data stored by any variable as if it belonged to any type. For example, the following code assigns data stored in a real variable to an integer variable.

```
type
  PInteger = ^Integer;
var
  R: Single;
  I: Integer;
  P: Pointer;
  PI: PInteger;
begin
  ...
  P := @R;
  PI := PInteger(P);
  I := PI^;
end;
```

Of course, reals and integers are stored in different formats. This assignment simply copies raw binary data from `R` to `I`, without converting it.

In addition to assigning the result of an @ operation, you can use several standard routines to give a value to a pointer. The `New` and `GetMem` procedures assign a memory address to an existing pointer, while the `Addr` and `Ptr` functions return a pointer to a specified address or variable.

Dereferenced pointers can be qualified and can function as qualifiers, as in the expression `P1^.Data^`.

The reserved word **nil** is a special constant that can be assigned to any pointer. When **nil** is assigned to a pointer, the pointer doesn't reference anything.

Pointer Types

You can declare a pointer to any type, using the syntax

type *pointerTypeName = ^type*

When you define a record or other data type, it might be useful to also to define a pointer to that type. This makes it easy to manipulate instances of the type without copying large blocks of memory.

Note: You can declare a pointer type before you declare the type it points to.

Standard pointer types exist for many purposes. The most versatile is **Pointer**, which can point to data of any kind. But a **Pointer** variable cannot be dereferenced; placing the **^** symbol after a **Pointer** variable causes a compilation error. To access the data referenced by a **Pointer** variable, first cast it to another pointer type and then dereference it.

Character Pointers

The fundamental types **PAnsiChar** and **PWideChar** represent pointers to **AnsiChar** and **WideChar** values, respectively. The generic **PChar** represents a pointer to a **Char** (that is, in its current implementation, to an **AnsiChar**). These character pointers are used to manipulate null-terminated strings. (See Working with null-terminated strings (see page 561).)

Type-checked Pointers

The `$T` compiler directive controls the types of pointer values generated by the @ operator. This directive takes the form of:

```
{$T+} or {$T-}
```

In the `{$T-}` state, the result type of the @ operator is always an untyped pointer that is compatible with all other pointer types. When @ is applied to a variable reference in the `{$T+}` state, the type of the result is `^T`, where `T` is compatible only with pointers to the type of the variable.

Other Standard Pointer Types

The `System` and `SysUtils` units declare many standard pointer types that are commonly used.

Selected pointer types declared in System and SysUtils

Pointer type	Points to variables of type
PAnsiString, **PString**	**AnsiString**
PByteArray	**TByteArray** (declared in `SysUtils`). Used to typecast dynamically allocated memory for array access.
PCurrency, **PDouble**, **PExtended**, **PSingle**	**Currency**, **Double**, **Extended**, **Single**
PInteger	**Integer**
POleVariant	**OleVariant**
PShortString	**ShortString**. Useful when porting legacy code that uses the old **PString** type.
PTextBuf	**TTextBuf** (declared in `SysUtils`). **TTextBuf** is the internal buffer type in a **TTextRec** file record.)
PVarRec	**TVarRec** (declared in `System`)
PVariant	**Variant**
PWideString	**WideString**
PWordArray	**TWordArray** (declared in `SysUtils`). Used to typecast dynamically allocated memory for arrays of 2-byte values.

See Also

Data Types (see page 553)

Simple Types (see page 554)

String Types (see page 561)

Structured Types (see page 566)

Procedural Types (see page 578)

Variant Types (see page 580)

Type Compatibility and Identity (see page 583)

Declaring Types (see page 586)

Variables (see page 587)

Declared Constants (see page 589)

3.1.3.2.6 Procedural Types

Procedural types allow you to treat procedures and functions as values that can be assigned to variables or passed to other procedures and functions.

About Procedural Types

Procedural types allow you to treat procedures and functions as values that can be assigned to variables or passed to other procedures and functions. For example, suppose you define a function called `Calc` that takes two integer parameters and returns an integer:

```
function Calc(X,Y: Integer): Integer;
```

You can assign the `Calc` function to the variable `F`:

```
var F: function(X,Y: Integer): Integer;
F := Calc;
```

If you take any procedure or function heading and remove the identifier after the word **procedure** or **function**, what's left is the name of a procedural type. You can use such type names directly in variable declarations (as in the previous example) or to declare new types:

```
type
  TIntegerFunction = function: Integer;
  TProcedure = procedure;
  TStrProc = procedure(const S: string);
  TMathFunc = function(X: Double): Double;
var
  F: TIntegerFunction;   { F is a parameterless function that returns an integer }
  Proc: TProcedure;      { Proc is a parameterless procedure }
  SP: TStrProc;          { SP is a procedure that takes a string parameter }
  M: TMathFunc;          { M is a function that takes a Double (real) parameter and returns a
Double }

  procedure FuncProc(P: TIntegerFunction);  { FuncProc is a procedure whose only parameter is
a parameterless integer-valued function }
```

On Win32, the variables shown in the previous example are all procedure pointers - that is, pointers to the address of a procedure or function. On the .NET platform, procedural types are implemented as delegates. If you want to reference a method of an instance object (see Classes and objects (see page 514)), you need to add the words `of object` to the procedural type name. For example

```
type
  TMethod      = procedure of object;
  TNotifyEvent = procedure(Sender: TObject) of object;
```

These types represent method pointers. A method pointer is really a pair of pointers; the first stores the address of a method, and the second stores a reference to the object the method belongs to. Given the declarations

```
type
  TNotifyEvent = procedure(Sender: TObject) of object;
  TMainForm = class(TForm)
    procedure ButtonClick(Sender: TObject);
     ...
  end;
var
  MainForm: TMainForm;
  OnClick: TNotifyEvent
```

we could make the following assignment.

```
OnClick := MainForm.ButtonClick;
```

Two procedural types are compatible if they have

- the same calling convention,
- the same return value (or no return value), and
- the same number of parameters, with identically typed parameters in corresponding positions. (Parameter names do not matter.)

On Win32, procedure pointer types are always incompatible with method pointer types, but this is not true on the .NET platform. The value **nil** can be assigned to any procedural type.

Nested procedures and functions (routines declared within other routines) cannot be used as procedural values, nor can predefined procedures and functions. If you want to use a predefined routine like `Length` as a procedural value, write a wrapper for it:

```
function FLength(S: string): Integer;
          begin
            Result := Length(S);
          end;
```

Procedural Types in Statements and Expressions

When a procedural variable is on the left side of an assignment statement, the compiler expects a procedural value on the right. The assignment makes the variable on the left a pointer to the function or procedure indicated on the right. In other contexts, however, using a procedural variable results in a call to the referenced procedure or function. You can even use a procedural variable to pass parameters:

```
var
  F: function(X: Integer): Integer;
  I: Integer;
  function SomeFunction(X: Integer): Integer;
    ...
  F := SomeFunction;    // assign SomeFunction to F
  I := F(4);            // call function; assign result to I
```

In assignment statements, the type of the variable on the left determines the interpretation of procedure or method pointers on the right. For example,

```
var
  F, G: function: Integer;
  I: Integer;
  function SomeFunction: Integer;
    ...
  F := SomeFunction;      // assign SomeFunction to F
  G := F;                 // copy F to G
  I := G;                 // call function; assign result to I
```

The first statement assigns a procedural value to F. The second statement copies that value to another variable. The third statement makes a call to the referenced function and assigns the result to `I`. Because `I` is an integer variable, not a procedural one, the last assignment actually calls the function (which returns an integer).

In some situations it is less clear how a procedural variable should be interpreted. Consider the statement

```
if F = MyFunction then ...;
```

In this case, the occurrence of F results in a function call; the compiler calls the function pointed to by F, then calls the function `MyFunction`, then compares the results. The rule is that whenever a procedural variable occurs within an expression, it represents a call to the referenced procedure or function. In a case where F references a procedure (which doesn't return a value), or where F references a function that requires parameters, the previous statement causes a compilation error. To compare the procedural value of F with `MyFunction`, use

```
if @F = @MyFunction then ...;
```

@F converts F into an untyped pointer variable that contains an address, and `@MyFunction` returns the address of `MyFunction`.

To get the memory address of a procedural variable (rather than the address stored in it), use @@. For example, @@F returns the address of F.

The @ operator can also be used to assign an untyped pointer value to a procedural variable. For example,

```
var StrComp: function(Str1, Str2: PChar): Integer;
   ...
@StrComp := GetProcAddress(KernelHandle, 'lstrcmpi');
```

calls the `GetProcAddress` function and points `StrComp` to the result.

Any procedural variable can hold the value **nil**, which means that it points to nothing. But attempting to call a nil-valued procedural variable is an error. To test whether a procedural variable is assigned, use the standard function `Assigned`:

```
if Assigned(OnClick) then OnClick(X);
```

See Also

Data Types (↗ see page 553)

Simple Types (↗ see page 554)

String Types (↗ see page 561)

Structured Types (↗ see page 566)

Pointers and Pointer Types (↗ see page 575)

Variant Types (↗ see page 580)

Type Compatibility and Identity (↗ see page 583)

Declaring Types (↗ see page 586)

Variables (↗ see page 587)

Declared Constants (↗ see page 589)

3.1.3.2.7 Variant Types

This topic discusses the use of variant data types.

Variants Overview

Sometimes it is necessary to manipulate data whose type varies or cannot be determined at compile time. In these cases, one option is to use variables and parameters of type Variant, which represent values that can change type at runtime. Variants offer greater flexibility but consume more memory than regular variables, and operations on them are slower than on statically bound types. Moreover, illicit operations on variants often result in runtime errors, where similar mistakes with regular variables would have been caught at compile time. You can also create custom variant types.

By default, Variants can hold values of any type except records, sets, static arrays, files, classes, class references, and pointers. In other words, variants can hold anything but structured types and pointers. They can hold interfaces, whose methods and properties can be accessed through them. (See Object interfaces (↗ see page 625).) They can hold dynamic arrays, and they can hold a special kind of static array called a variant array. (See Variant arrays.) Variants can mix with other variants and with integer, real, string, and **Boolean** values in expressions and assignments; the compiler automatically performs type conversions.

Variants that contain strings cannot be indexed. That is, if V is a variant that holds a string value, the construction `V[1]` causes a runtime error.

You can define custom Variants that extend the Variant type to hold arbitrary values. For example, you can define a Variant string type that allows indexing or that holds a particular class reference, record type, or static array. Custom Variant types are defined by creating descendants to the `TCustomVariantType` class.

Note: This, and almost all variant functionality, is implemented in the `Variants` unit.

A variant occupies 16 bytes of memory and consists of a type code and a value, or pointer to a value, of the type specified by the code. All variants are initialized on creation to the special value Unassigned. The special value `Null` indicates unknown or missing data.

The standard function `VarType` returns a variant's type code. The `varTypeMask` constant is a bit mask used to extract the code from `VarType`'s return value, so that, for example,

```
VarType(V) and varTypeMask = varDouble
```

returns **True** if `V` contains a **Double** or an array of **Double**. (The mask simply hides the first bit, which indicates whether the variant holds an array.) The TVarData record type defined in the `System` unit can be used to typecast variants and gain access to their internal representation.

Variant Type Conversions

All integer, real, string, character, and Boolean types are assignment-compatible with Variant. Expressions can be explicitly cast as variants, and the `VarAsType` and `VarCast` standard routines can be used to change the internal representation of a variant. The following code demonstrates the use of variants and some of the automatic conversions performed when variants are mixed with other types.

```
var
   V1, V2, V3, V4, V5: Variant;
   I: Integer;
   D: Double;
   S: string;
   begin
      V1 := 1;         { integer value }
      V2 := 1234.5678; { real value }
      V3 := 'Hello world!'; { string value }
      V4 := '1000';         { string value }
      V5 := V1 + V2 + V4;   { real value 2235.5678}
      I  := V1;             { I = 1 (integer value) }
      D  := V2;              { D = 1234.5678 (real value) }
      S  := V3;              { S = 'Hello world!' (string value) }
      I  := V4;              { I = 1000 (integer value) }
      S  := V5;              { S = '2235.5678' (string value) }
   end;
```

The compiler performs type conversions according to the following rules.

Variant type conversion rules

Target **Source**	**integer**	**real**	**string**	**Boolean**
integer	converts integer formats	converts to real	converts to string representation	returns **False** if 0, True otherwise
real	rounds to nearest integer	converts real formats	converts to string representation using regional settings	returns **False** if 0, **True** otherwise

string	converts to integer, truncating if necessary; raises exception if string is not numeric	converts to real using regional settings; raises exception if string is not numeric	converts string/character formats	returns **False** if string is 'false' (noncase-sensitive) or a numeric string that evaluates to 0, **True** if string is 'true' or a nonzero numeric string; raises exception otherwise
character	same as string (above)	same as string (above)	same as string (above)	same as string (above)
Boolean	**False** = 0, **True**: all bits set to 1 (-1 if Integer, 255 if Byte, etc.)	**False** = 0, True = 1	**False** = 'False', **True** = 'True' by default; casing depends on global variable BooleanToStringRule	**False** = **False**, **True** = **True**
Unassigned	returns 0	returns 0	returns empty string	returns **False**
Null	depends on global variable NullStrictConvert (raises an exception by default)	depends on global variable NullStrictConvert (raises an exception by default)	depends on global variables NullStrictConvert and NullAsStringValue (raises an exception by default)	depends on global variable NullStrictConvert (raises an exception by default)

Out-of-range assignments often result in the target variable getting the highest value in its range. Invalid variant operations, assignments or casts raise an EVariantError exception or an exception class decending from EVariantError.

Special conversion rules apply to the TDateTime type declared in the `System` unit. When a TDateTime is converted to any other type, it treated as a normal **Double**. When an integer, real, or Boolean is converted to a TDateTime, it is first converted to a **Double**, then read as a date-time value. When a string is converted to a TDateTime, it is interpreted as a date-time value using the regional settings. When an `Unassigned` value is converted to TDateTime, it is treated like the real or integer value 0. Converting a `Null` value to TDateTime raises an exception.

On the Win32 platform, if a variant references a COM interface, any attempt to convert it reads the object's default property and converts that value to the requested type. If the object has no default property, an exception is raised.

Variants in Expressions

All operators except **^**, **is**, and **in** take variant operands. Except for comparisons, which always return a **Boolean** result, any operation on a variant value returns a variant result. If an expression combines variants with statically-typed values, the statically-typed values are automatically converted to variants.

This is not true for comparisons, where any operation on a Null variant produces a `Null` variant. For example:

```
V := Null + 3;
```

assigns a `Null` variant to V. By default, comparisons treat the `Null` variant as a unique value that is less than any other value. For example:

```
if Null > -3 then ... else ...;
```

In this example, the **else** part of the **if** statement will be executed. This behavior can be changed by setting the `NullEqualityRule` and `NullMagnitudeRule` global variables.

Variant Arrays

You cannot assign an ordinary static array to a variant. Instead, create a variant array by calling either of the standard functions `VarArrayCreate` or `VarArrayOf`. For example,

```
V: Variant;
   ...
V := VarArrayCreate([0,9], varInteger);
```

creates a variant array of integers (of length 10) and assigns it to the variant `V`. The array can be indexed using `V[0]`, `V[1]`, and so forth, but it is not possible to pass a variant array element as a **var** parameter. Variant arrays are always indexed with integers.

The second parameter in the call to `VarArrayCreate` is the type code for the array's base type. For a list of these codes, see VarType. Never pass the code `varString` to `VarArrayCreate`; to create a variant array of strings, use `varOleStr`.

Variants can hold variant arrays of different sizes, dimensions, and base types. The elements of a variant array can be of any type allowed in variants except **ShortString** and **AnsiString**, and if the base type of the array is **Variant**, its elements can even be heterogeneous. Use the `VarArrayRedim` function to resize a variant array. Other standard routines that operate on variant arrays include `VarArrayDimCount`, `VarArrayLowBound`, `VarArrayHighBound`, `VarArrayRef`, `VarArrayLock`, and `VarArrayUnlock`.

Note: Variant arrays of custom variants are not supported, as instances of custom variants can be added to a `VarVariant` variant array.

When a variant containing a variant array is assigned to another variant or passed as a value parameter, the entire array is copied. Don't perform such operations unnecessarily, since they are memory-inefficient.

OleVariant

The **OleVariant** type exists on both the Windows and Linux platforms. The main difference between *Variant* and **OleVariant** is that *Variant* can contain data types that only the current application knows what to do with. **OleVariant** can only contain the data types defined as compatible with OLE Automation which means that the data types that can be passed between programs or across the network without worrying about whether the other end will know how to handle the data.

When you assign a *Variant* that contains custom data (such as a Delphi string, or a one of the new custom variant types) to an **OleVariant**, the runtime library tries to convert the *Variant* into one of the **OleVariant** standard data types (such as a Delphi string converts to an OLE BSTR string). For example, if a variant containing an **AnsiString** is assigned to an **OleVariant**, the **AnsiString** becomes a **WideString**. The same is true when passing a *Variant* to an **OleVariant** function parameter.

See Also

Data Types (see page 553)

Simple Types (see page 554)

String Types (see page 561)

Structured Types (see page 566)

Pointers and Pointer Types (see page 575)

Procedural Types (see page 578)

Type Compatibility and Identity (see page 583)

Declaring Types (see page 586)

Variables (see page 587)

Declared Constants (see page 589)

3.1.3.2.8 Type Compatibility and Identity

To understand which operations can be performed on which expressions, we need to distinguish several kinds of compatibility among types and values. These include:

- Type identity
- Type compatibility
- Assignment compatibility

Type Identity

When one type identifier is declared using another type identifier, without qualification, they denote the same type. Thus, given the declarations

```
type
  T1 = Integer;
  T2 = T1;
  T3 = Integer;
  T4 = T2;
```

`T1`, `T2`, `T3`, `T4`, and **Integer** all denote the same type. To create distinct types, repeat the word **type** in the declaration. For example,

```
type TMyInteger = type Integer;
```

creates a new type called `TMyInteger` which is not identical to Integer.

Language constructions that function as type names denote a different type each time they occur. Thus the declarations

```
type
  TS1 = set of Char;
  TS2 = set of Char;
```

create two distinct types, `TS1` and `TS2`. Similarly, the variable declarations

```
var
  S1: string[10];
  S2: string[10];
```

create two variables of distinct types. To create variables of the same type, use

```
var S1, S2: string[10];
```

or

```
type MyString = string[10];
var
    S1: MyString;
    S2: MyString;
```

Type Compatibility

Every type is compatible with itself. Two distinct types are compatible if they satisfy at least one of the following conditions.

- They are both real types.
- They are both integer types.
- One type is a subrange of the other.
- Both types are subranges of the same type.
- Both are set types with compatible base types.
- Both are packed-string types with the same number of characters.
- One is a string type and the other is a string, packed-string, or **Char** type.
- One type is Variant and the other is an integer, real, string, character, or Boolean type.
- Both are class, class-reference, or interface types, and one type is derived from the other.
- One type is **PChar** or **PWideChar** and the other is a zero-based character array of the form `array[0..n]` of **PChar** or **PWideChar**.
- One type is **Pointer** (an untyped pointer) and the other is any pointer type.
- Both types are (typed) pointers to the same type and the `{$T+}` compiler directive is in effect.
- Both are procedural types with the same result type, the same number of parameters, and type-identity between parameters in corresponding positions.

Assignment Compatibility

Assignment-compatibility is not a symmetric relation. An expression of type T2 can be assigned to a variable of type T1 if the value of the expression falls in the range of T1 and at least one of the following conditions is satisfied.

- T1 and T2 are of the same type, and it is not a file type or structured type that contains a file type at any level.
- T1 and T2 are compatible ordinal types.
- T1 and T2 are both real types.
- T1 is a real type and T2 is an integer type.
- T1 is **PChar**, **PWideChar** or any string type and the expression is a string constant.
- T1 and T2 are both string types.
- T1 is a string type and T2 is a **Char** or packed-string type.
- T1 is a long string and T2 is **PChar** or **PWideChar**.
- T1 and T2 are compatible packed-string types.
- T1 and T2 are compatible set types.
- T1 and T2 are compatible pointer types.
- T1 and T2 are both class, class-reference, or interface types and T2 is a derived from T1.
- T1 is an interface type and T2 is a class type that implements T1.
- T1 is **PChar** or **PWideChar** and T2 is a zero-based character array of the form `array[0..n] of Char` (when T1 is **PChar**) or of **WideChar** (when T1 is **PWideChar**).
- T1 and T2 are compatible procedural types. (A function or procedure identifier is treated, in certain assignment statements, as an expression of a procedural type.)
- T1 is Variant and T2 is an integer, real, string, character, **Boolean**, interface type or **OleVariant** type.
- T1 is an **OleVariant** and T2 is an integer, real, string, character, **Boolean**, interface, or Variant type.
- T1 is an integer, real, string, character, or **Boolean** type and T2 is Variant or **OleVariant**.
- T1 is the `IUnknown` or `IDispatch` interface type and T2 is Variant or **OleVariant**. (The variant's type code must be `varEmpty`, `varUnknown`, or `varDispatch` if `T1` is `IUnknown`, and `varEmpty` or `varDispatch` if `T1` is `IDispatch`.)

See Also

Data Types (↗ see page 553)

Simple Types (↗ see page 554)

String Types (↗ see page 561)

Structured Types (↗ see page 566)

Pointers and Pointer Types (↗ see page 575)

Procedural Types (↗ see page 578)

Variant Types (↗ see page 580)

Declaring Types (↗ see page 586)

Variables (↗ see page 587)

Declared Constants (↗ see page 589)

Overview of Generics (Parameterized Types) (↗ see page 596)

Declaring Generics (Parameterized Types) (↗ see page 598)

Overloads and Type Compatibility (Parameterized Types)

3.1.3.2.9 Declaring Types

This topic describes the syntax of Delphi type declarations.

Type Declaration Syntax

A type declaration specifies an identifier that denotes a type. The syntax for a type declaration is

type *newTypeName* = *type*

where *newTypeName* is a valid identifier. For example, given the type declaration

```
type TMyString = string;
```

you can make the variable declaration

```
var S: TMyString;
```

A type identifier's scope doesn't include the type declaration itself (except for pointer types). So you cannot, for example, define a record type that uses itself recursively.

When you declare a type that is identical to an existing type, the compiler treats the new type identifier as an alias for the old one. Thus, given the declarations

```
type TValue = Real;
var
  X: Real;
  Y: TValue;
```

X and Y are of the same type; at runtime, there is no way to distinguish `TValue` from **Real**. This is usually of little consequence, but if your purpose in defining a new type is to utilize runtime type information, for example, to associate a property editor with properties of a particular type - the distinction between 'different name' and 'different type' becomes important. In this case, use the syntax

`type` *newTypeName* = **type**_type_

For example,

```
type TValue = type Real;
```

forces the compiler to create a new, distinct type called `TValue`.

For `var` parameters, types of formal and actual must be identical. For example,

```
type
  TMyType = type Integer
  procedure p(var t:TMyType);
  begin
  end;

procedure x;
var
  m: TMyType;
  i: Integer;
begin
  p(m); // Works
  p(i); // Error! Types of formal and actual must be identical.
end;
```

Note: This only applies to `var` parameters, not to `const` or by-value parameters.

See Also

Data Types (see page 553)

Simple Types (see page 554)

String Types (see page 561)

Structured Types (see page 566)

Pointers and Pointer Types (see page 575)

Procedural Types (see page 578)

Variant Types (see page 580)

Type Compatibility and Identity (see page 583)

Variables (see page 587)

Declared Constants (see page 589)

3.1.3.2.10 Variables

A variable is an identifier whose value can change at runtime. Put differently, a variable is a name for a location in memory; you can use the name to read or write to the memory location. Variables are like containers for data, and, because they are typed, they tell the compiler how to interpret the data they hold.

Declaring Variables

The basic syntax for a variable declaration is

var *identifierList*:*type;*

where *identifierList* is a comma-delimited list of valid identifiers and type is any valid type. For example,

```
var I: Integer;
```

declares a variable `I` of type **Integer**, while

```
var X, Y: Real;
```

declares two variables - `X` and `Y` - of type **Real**.

Consecutive variable declarations do not have to repeat the reserved word **var**:

```
var
   X, Y, Z: Double;
   I, J, K: Integer;
   Digit: 0..9;
   Okay: Boolean;
```

Variables declared within a procedure or function are sometimes called local, while other variables are called global. Global variables can be initialized at the same time they are declared, using the syntax

var *identifier: type* = *constantExpression;*

where *constantExpression* is any constant expression representing a value of type *type*. Thus the declaration

```
var I: Integer = 7;
```

is equivalent to the declaration and statement

```
var I: Integer;
    ...
I := 7;
```

Local variables cannot be initialized in their declarations. Multiple variable declarations (such as `var X, Y, Z: Real;`) cannot include initializations, nor can declarations of variant and file-type variables.

If you don't explicitly initialize a global variable, the compiler initializes it to 0. Object instance data (fields) are also initialized to 0.

On the Wiin32 platform, the contents of a local variable are undefined until a value is assigned to them. On the .NET platform, the CLR initializes all variables, including local variables, to 0.

When you declare a variable, you are allocating memory which is freed automatically when the variable is no longer used. In particular, local variables exist only until the program exits from the function or procedure in which they are declared. For more information about variables and memory management, see Memory management (see page 644).

Absolute Addresses

You can create a new variable that resides at the same address as another variable. To do so, put the directive **absolute** after the type name in the declaration of the new variable, followed by the name of an existing (previously declared) variable. For example,

```
var
   Str: string[32];
   StrLen: Byte absolute Str;
```

specifies that the variable `StrLen` should start at the same address as `Str`. Since the first byte of a short string contains the string's length, the value of `StrLen` is the length of `Str`.

You cannot initialize a variable in an **absolute** declaration or combine **absolute** with any other directives.

Dynamic Variables

You can create dynamic variables by calling the `GetMem` or `New` procedure. Such variables are allocated on the heap and are not managed automatically. Once you create one, it is your responsibility ultimately to free the variable's memory; use `FreeMem` to destroy variables created by `GetMem` and `Dispose` to destroy variables created by `New`. Other standard routines that operate on dynamic variables include `ReallocMem`, `AllocMem`, `Initialize`, `Finalize`, `StrAlloc`, and `StrDispose`.

Long strings, wide strings, dynamic arrays, variants, and interfaces are also heap-allocated dynamic variables, but their memory is managed automatically.

Thread-local Variables

Thread-local (or thread) variables are used in multithreaded applications. A thread-local variable is like a global variable, except that each thread of execution gets its own private copy of the variable, which cannot be accessed from other threads. Thread-local variables are declared with **threadvar** instead of **var**. For example,

```
threadvar X: Integer;
```

Thread-variable declarations

- cannot occur within a procedure or function.
- cannot include initializations.
- cannot specify the **absolute** directive.

Dynamic variables that are ordinarily managed by the compiler (long strings, wide strings, dynamic arrays, variants, and interfaces) can be declared with **threadvar**, but the compiler does not automatically free the heap-allocated memory created by each thread of execution. If you use these data types in thread variables, it is your responsibility to dispose of their memory from within the thread, before the thread terminates. For example,

```
threadvar S: AnsiString;
S := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
  ...
S := '';  // free the memory used by S
```

Note: Use of such constructs is discouraged.

You can free a variant by setting it to `Unassigned` and an interface or dynamic array by setting it to **nil**.

See Also

Data Types (see page 553)

Simple Types (↗ see page 554)

String Types (↗ see page 561)

Structured Types (↗ see page 566)

Pointers and Pointer Types (↗ see page 575)

Procedural Types (↗ see page 578)

Variant Types (↗ see page 580)

Type Compatibility and Identity (↗ see page 583)

Declaring Types (↗ see page 586)

Declared Constants (↗ see page 589)

3.1.3.2.11 Declared Constants

Several different language constructions are referred to as 'constants'. There are numeric constants (also called numerals) like 17, and string constants (also called character strings or string literals) like 'Hello world!'. Every enumerated type defines constants that represent the values of that type. There are predefined constants like **True**, **False**, and **nil**. Finally, there are constants that, like variables, are created individually by declaration.

Declared constants are either *true constants* or *typed constants*. These two kinds of constant are superficially similar, but they are governed by different rules and used for different purposes.

True Constants

A true constant is a declared identifier whose value cannot change. For example,

```
const MaxValue = 237;
```

declares a constant called `MaxValue` that returns the integer 237. The syntax for declaring a true constant is

const *identifier* = *constantExpression*

where identifier is any valid identifier and *constantExpression* is an expression that the compiler can evaluate without executing your program.

If *constantExpression* returns an ordinal value, you can specify the type of the declared constant using a value typecast. For example

```
const MyNumber = Int64(17);
```

declares a constant called `MyNumber`, of type **Int64**, that returns the integer 17. Otherwise, the type of the declared constant is the type of the *constantExpression*.

- If *constantExpression* is a character string, the declared constant is compatible with any string type. If the character string is of length 1, it is also compatible with any character type.
- If *constantExpression* is a real, its type is **Extended**. If it is an integer, its type is given by the table below.

Types for integer constants

Range of constant(hexadecimal)	Range of constant(decimal)	Type
-$8000000000000000..-$80000001	-2^63..-2147483649	**Int64**
-$80000000..-$8001	-2147483648..-32769	**Integer**
-$8000..-$81	-32768..-129	**Smallint**
-$80..-1	-128..-1	**Shortint**

0..$7F	0..127	0..127
$80..$FF	128..255	**Byte**
$0100..$7FFF	256..32767	0..32767
$8000..$FFFF	32768..65535	**Word**
$10000..$7FFFFFFF	65536..2147483647	0..2147483647
$80000000..$FFFFFFFF	2147483648..4294967295	**Cardinal**
$100000000..$7FFFFFFFFFFFFFFF	4294967296..2^63–1	**Int64**

Here are some examples of constant declarations:

```
const
  Min = 0;
  Max = 100;
  Center = (Max - Min) div 2;
  Beta = Chr(225);
  NumChars = Ord('Z') - Ord('A') + 1;
  Message = 'Out of memory';
  ErrStr = ' Error: ' + Message + '. ';
  ErrPos = 80 - Length(ErrStr) div 2;
  Ln10 = 2.302585092994045684;
  Ln10R = 1 / Ln10;
  Numeric = ['0'..'9'];
  Alpha = ['A'..'Z', 'a'..'z'];
  AlphaNum = Alpha + Numeric;
```

Constant Expressions

A constant expression is an expression that the compiler can evaluate without executing the program in which it occurs. Constant expressions include numerals; character strings; true constants; values of enumerated types; the special constants **True**, **False**, and **nil**; and expressions built exclusively from these elements with operators, typecasts, and set constructors. Constant expressions cannot include variables, pointers, or function calls, except calls to the following predefined functions:

`Abs`	`High`	`Low`	`Pred`	`Succ`
`Chr`	`Length`	`Odd`	`Round`	`Swap`
`Hi`	`Lo`	`Ord`	`SizeOf`	`Trunc`

This definition of a constant expression is used in several places in Delphi's syntax specification. Constant expressions are required for initializing global variables, defining subrange types, assigning ordinalities to values in enumerated types, specifying default parameter values, writing **case** statements, and declaring both true and typed constants.

Examples of constant expressions:

```
100
'A'
256 - 1
(2.5 + 1) / (2.5 - 1)
'Borland' + ' ' + 'Developer'
Chr(32)
Ord('Z') - Ord('A') + 1
```

Resource Strings

Resource strings are stored as resources and linked into the executable or library so that they can be modified without recompiling the program.

Resource strings are declared like other true constants, except that the word **const** is replaced by **resourcestring**. The expression to the right of the **=** symbol must be a constant expression and must return a string value. For example,

```
resourcestring
  CreateError = 'Cannot create file %s';
  OpenError = 'Cannot open file %s';
  LineTooLong = 'Line too long';
  ProductName = 'CodeGear Rocks';
  SomeResourceString = SomeTrueConstant;
```

Typed Constants

Typed constants, unlike true constants, can hold values of array, record, procedural, and pointer types. Typed constants cannot occur in constant expressions.

Declare a typed constant like this:

const *identifier: type = value*

where identifier is any valid identifier, type is any type except files and variants, and value is an expression of type. For example,

```
const Max: Integer = 100;
```

In most cases, value must be a constant expression; but if type is an array, record, procedural, or pointer type, special rules apply.

Array Constants

To declare an array constant, enclose the values of the array's elements, separated by commas, in parentheses at the end of the declaration. These values must be represented by constant expressions. For example,

```
const Digits: array[0..9] of Char = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');
```

declares a typed constant called `Digits` that holds an array of characters.

Zero-based character arrays often represent null-terminated strings, and for this reason string constants can be used to initialize character arrays. So the previous declaration can be more conveniently represented as

```
const Digits: array[0..9] of Char = '0123456789';
```

To define a multidimensional array constant, enclose the values of each dimension in a separate set of parentheses, separated by commas. For example,

```
type TCube = array[0..1, 0..1, 0..1] of Integer;
const Maze: TCube = (((0, 1), (2, 3)), ((4, 5), (6,7)));
```

creates an array called `Maze` where

```
Maze[0,0,0] = 0
Maze[0,0,1] = 1
Maze[0,1,0] = 2
Maze[0,1,1] = 3
Maze[1,0,0] = 4
Maze[1,0,1] = 5
Maze[1,1,0] = 6
Maze[1,1,1] = 7
```

Array constants cannot contain file-type values at any level.

Record Constants

To declare a record constant, specify the value of each field - as `fieldName: value`, with the field assignments separated by semicolons - in parentheses at the end of the declaration. The values must be represented by constant expressions. The fields must be listed in the order in which they appear in the record type declaration, and the tag field, if there is one, must have a value specified; if the record has a variant part, only the variant selected by the tag field can be assigned values.

Examples:

```
type
  TPoint = record
     X, Y: Single;
  end;
  TVector = array[0..1] of TPoint;
  TMonth = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
  TDate = record
    D: 1..31;
    M: TMonth;
    Y: 1900..1999;
  end;
const
  Origin: TPoint = (X: 0.0; Y: 0.0);
  Line: TVector = ((X: -3.1; Y: 1.5), (X: 5.8; Y: 3.0));
  SomeDay: TDate = (D: 2; M: Dec; Y: 1960);
```

Record constants cannot contain file-type values at any level.

Procedural Constants

To declare a procedural constant, specify the name of a function or procedure that is compatible with the declared type of the constant. For example,

```
function Calc(X, Y: Integer): Integer;
begin
  ...
end;

type TFunction = function(X, Y: Integer): Integer;
const MyFunction: TFunction = Calc;
```

Given these declarations, you can use the procedural constant `MyFunction` in a function call:

```
I := MyFunction(5, 7)
```

You can also assign the value **nil** to a procedural constant.

Pointer Constants

When you declare a pointer constant, you must initialize it to a value that can be resolved at least as a relative address at compile time. There are three ways to do this: with the @ operator, with **nil**, and (if the constant is of type **PChar** or **PWideChar**) with a string literal. For example, if `I` is a global variable of type **Integer**, you can declare a constant like

```
const PI: ^Integer = @I;
```

The compiler can resolve this because global variables are part of the code segment. So are functions and global constants:

```
const PF: Pointer = @MyFunction;
```

Because string literals are allocated as global constants, you can initialize a **PChar** constant with a string literal:

```
const WarningStr: PChar = 'Warning!';
```

See Also

Data Types (↗ see page 553)

Simple Types (↗ see page 554)

String Types (↗ see page 561)

Structured Types (↗ see page 566)

Pointers and Pointer Types (↗ see page 575)

Procedural Types (↗ see page 578)

Variant Types (↗ see page 580)

Type Compatibility and Identity (↗ see page 583)

Declaring Types (↗ see page 586)

Variables (↗ see page 587)

3.1.3.3 .NET Topics

This section contains information specific to programming in Delphi on the .NET platform.

Topics

Name	Description
Using .NET Custom Attributes (↗ see page 593)	.NET framework assemblies are self-describing entities. They contain intermediate code that is compiled to native machine instructions when the assembly is loaded. More than that, assemblies contain a wealth of information about that code. The compiler emits this descriptive information, or metadata, into the assembly as it processes the source code. In other programming environments, there is no way to access metadata once your code is compiled; the information is lost during the compilation process. On the .NET platform, however, you have the ability to access metadata using runtime reflection services. The .NET framework gives you the ability to extend... more (↗ see page 593)

3.1.3.3.1 Using .NET Custom Attributes

.NET framework assemblies are self-describing entities. They contain intermediate code that is compiled to native machine instructions when the assembly is loaded. More than that, assemblies contain a wealth of information about that code. The compiler emits this descriptive information, or metadata, into the assembly as it processes the source code. In other programming environments, there is no way to access metadata once your code is compiled; the information is lost during the compilation process. On the .NET platform, however, you have the ability to access metadata using runtime reflection services.

The .NET framework gives you the ability to extend the metadata emitted by the compiler with your own descriptive attributes. These customized attributes are somewhat analogous to language keywords, and are stored with the other metadata in the assembly.

- Declaring custom attributes
- Using custom attributes
- Custom attributes and interfaces

Declaring a Custom Attribute Class

Creating a custom attribute is the same as declaring a class. The custom attribute class has a constructor, and properties to set and retrieve its state data. Custom attributes must inherit from TCustomAttribute. The following code declares a custom attribute with a constructor and two properties:

```
type
   TCustomCodeAttribute = class(TCustomAttribute)
      private
          Fprop1 : integer;
          Fprop2 : integer;
          aVal   : integer;
          procedure Setprop1(p1 : integer);
          procedure Setprop2(p2 : integer);
      public
          constructor Create(const myVal : integer);
          property prop1 : integer read Fprop1 write Setprop1;
```

```
              property prop2 : integer read Fprop2 write Setprop2;
       end;
```

The implementation of the constructor might look like

```
constructor TCustomCodeAttribute.Create(const myVal : integer);
begin
    inherited Create;
    aVal := myVal;
end;
```

Delphi for .NET supports the creation of custom attribute classes, as shown above, and all of the custom attributes provided by the .NET framework.

Using Custom Attributes

Custom attributes are placed directly before the source code symbol to which the attribute applies. Attributes can be placed before

- variables and constants
- procedures and functions
- function results
- procedure and function parameters
- types
- fields, properties, and methods

Note that Delphi for .NET supports the use of named properties in the initialization. These can be the names of properties, or of public fields of the custom attribute class. Named properties are listed after all of the parameters required by the constructor. For example

```
[TCustomCodeAttribute(1024, prop1=512, prop2=128)]
TMyClass = class(TObject)
...
end;
```

applies the custom attribute declared above to the class `TMyClass`.

The first parameter, 1024, is the value required by the constructor. The second two parameters are the properties defined in the custom attribute.

When a custom attribute is placed before a list of multiple variable declarations, the attribute applies to all variables declared in that list. For example

```
var
 [TCustomAttribute(1024, prop1=512, prop2=128)]
 x, y, z: Integer;
```

would result in `TCustomAttribute` being applied to all three variables, X, Y, and Z.

Custom attributes applied to types can be detected at runtime with the GetCustomAttributes method of the Type class. The following Delphi code demonstrates how to query for custom attributes at runtime.

```
var
   F: TMyClass;          // TMyClass declared above
   T: System.Type;
   A: array of TObject;  // Will hold custom attributes
   I: Integer;

begin
   F := TMyClass.Create;
   T := F.GetType;
   A := T.GetCustomAttributes(True);

   // Write the type name, and then loop over custom
```

```
    // attributes returned from the call to
    // System.Type.GetCustomAttributes.
    Writeln(T.FullName);
    for I := Low(A) to High(A) do
       Writeln(A[I].GetType.FullName);
end.
```

Using the DllImport Custom Attribute

You can call unmanaged Win32 APIs (and other unmanaged code) by prefixing the function declaration with the `DllImport` custom attribute. This attribute resides in the `System.Runtime.InteropServices` namespace, as shown below:

```
Program HellowWorld2;

    // Don't forget to include the InteropServices unit when using the DllImport attribute.
    uses System.Runtime.InteropServices;

    [DllImport('user32.dll')]
    function MessageBeep(uType : LongWord) : Boolean; external;

    begin
          MessageBeep(LongWord(-1));
    end.
```

Note the **external** keyword is still required, to replace the block in the function declaration. All other attributes, such as the calling convention, can be passed through the `DllImport` custom attribute.

Custom Attributes and Interfaces

Delphi syntax dictates that the GUID (if present) must immediately follow the declaration of an interface. Since the GUID syntax is similar to that of custom attributes, the compiler must be made to know the difference between a custom attribute - which applies to the next declaration - and a GUID specifier, which applies to the previous declaration. Without this special case, the compiler would try to apply an attribute to the first member of the interface.

When the compiler sees an interface declaration, the next square bracket construct found is assumed to be that of a GUID specifier for the interface. The GUID must be in the traditional Delphi form:

```
['{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}']
```

Alternatively, you can use the `Guid` custom attribute of the .NET framework (GuidAttribute). If you choose this method, then you should introduce the attribute before the interface, as with any other custom attribute.

The effect in either case is the same: the GUID is emitted into the metadata for the interface type. Note that GUIDs are not required for interfaces in the .NET Framework. They are only used for COM interoperability.

Note: When importing COM interfaces with the `ComImport` custom attribute, you must declare the `GuidAttribute` instead of using the Delphi syntax.

See Also

Using Platform Invoke with Delphi for .NET

3.1.3.4 Generics (Parameterized Types)

Presents an overview of generics, a terminology list, a summary of grammar changes for generics, and details about declaring and using parameterized types, specifying constraints on generics, and using overloads.

Topics

Name	Description
Overview of Generics (see page 596)	Delphi for .NET supports the use of generics, also known as parameterized types.
Terminology for Generics (see page 597)	

Declaring Generics (↗ see page 598)	The declaration of a generic is similar to the declaration of a regular class, record, or interface type. The difference is that a list of one or more type parameters placed between angle brackets (< and >) follows the type identifier in the declaration of a generic. Type parameters can be used as a typical type identifier inside the container type declaration and method body. For example:
Overloads and Type Compatibility in Generics (↗ see page 602)	
Constraints in Generics (↗ see page 603)	Constraints can be associated with a type parameter of a generic. Constraints declare items that must be supported by any concrete type passed to that parameter in a construction of the generic type.
Class Variable in Generics (↗ see page 605)	The class variable defined in a generic type is instantiated in each instantiated type identified by the type parameters. The following code shows that TFoo<Integer>.FCount and TFoo<String>.FCount are instantiated only once, and these are two different variables.
Changes in Standard Functions and Grammar (↗ see page 606)	Here is a list of standard function changes to support parameterized types. Example forms:

3.1.3.4.1 Overview of Generics

Delphi for .NET supports the use of generics, also known as parameterized types.

How Generics Work

The term **generics** is a collective noun that describes the set of things in the platform that can be parameterized by type using the new support in .NET 2.0. **Generics** can refer to generic types, generic methods, or (for Delphi) generic procedures and generic functions.

Generics are a set of abstraction tools that permit the decoupling of an algorithm (such as a method, procedure or function) or a data structure (such as a class, interface or record) from one or more concrete types that the algorithm or data structure uses.

A method or data type that uses other types in its definition can be made more general by substituting one or more concrete types with type parameters. Then you add those type parameters to a type parameter list in the method or data structure declaration. This is similar to the way that you can make a procedure more general by substituting instances of a literal constant in its body with a parameter name, and adding the parameter to the parameter list of the procedure.

For example, a class (TMyList) that maintains a list of objects (of type TObject) can be made more reusable and type-safe by substituting uses of TObject with a type parameter name (such as 'T'), and adding the type parameter to the class's type parameter list so that it becomes TMyList<T>.

Concrete uses (**instantiations**) of a generic type or method can be made by supplying type arguments to the generic type or method at the point of use. The act of supplying type arguments effectively constructs a new type or method by substituting instances of the type parameter in the generic definition with the corresponding type argument.

For example, the list might be used as TMyList<Double>. This creates a new type, TMyList<Double>, whose definition is identical to TMyList<T> except that all instances of 'T' in the definition are replaced with 'Double'.

It should be noted that generics as an abstraction mechanism duplicates much of the functionality of **polymorphism**, but with different characteristics. Since a new type or method is constructed at instantiation time, you can find type errors sooner, at compile time rather than run time. This also increases the scope for optimization, but with a trade-off - each instantiation increases the memory usage of the final running application, possibly resulting in lower performance.

Code Examples

For example, TSIPair is a class holding two data types, String and Integer:

```
type
  TSIPair = class
  private
    FKey: String;
    FValue: Integer;
  public
```

```
    function GetKey: String;
    procedure SetKey(Key: String);
    function GetValue: Integer;
    procedure SetValue(Value: Integer);
    property Key: TKey read GetKey write SetKey;
    property Value: TValue read GetValue write SetValue;
  end;
```

To make a class independent of data type, replace the data type with a type parameter.

```
type
  TPair<TKey,TValue>= class   // declares TPair type with two type parameters

private
    FKey: TKey;
    FValue: TValue;
  public
    function GetKey: TKey;
    procedure SetKey(Key: TKey);
    function GetValue: TValue;
    procedure SetValue(Value: TValue);
    property Key: TKey read GetKey write SetKey;
    property Value: TValue read GetValue write SetValue;
  end;

type
  TSIPair = TPair<String,Integer>; // declares instantiated type
  TSSPair = TPair<String,String>; // declares with other data types
  TISPair = TPair<Integer,String>;
    TIIPair = TPair<Integer,Integer>;
    TSXPair = TPair<String,TXMLNode>;
```

Platform Requirements and Differences

Parameterized types are supported by .NET 2.0 or later, but are not supported by .NET 1.1 or before.

Instantiation timing:

On .NET, instantiation is processed by the run time environment.

See Also

Terminology of Generics (see page 597)

Declaration of Generics (see page 598)

Overloads and Type Compatibility (Parameterized Types) (see page 602)

Constraints (Parameterized Types) (see page 603)

Class Variable in Parameterized Types (see page 605)

Changes in Standard Functions and Grammar (Parameterized Types) (see page 606)

3.1.3.4.2 Terminology for Generics

Terms Used in Describing Generics

Terminology used to describe generics (parameterized types) is defined in this section.

Type parameterized type	A type declaration which requires type parameters to be supplied in order to form an actual type.
	'List<Item>' is a type parameterized type in the following code:
	`type`
	`List<Item> = class`
	`...`
	`end;`

Parameterized type	Same as type parameterized type.

Type parameter	A parameter declared in a parameterized type declaration or a method header in order to use as a type for another declaration inside its parameterized type declaration or method body. It will be bound to real type argument. Item is a type parameter in the following code:
	`type`
	`List<Item> = class`
	`...`
	`end;`

Type argument	A type used with type identifier in order to make instantiated type. In List<Integer>, if List is a parameterized type, Integer is a type argument.
Instantiated type	The combination of a parameterized type with a set of parameters.
Constructed type	Same as instantiated typed.
Closed constructed type	A constructed type having all its parameters resolved to actual types. List<Integer> is closed because integer is an actual type.
Open constructed type	A constructed type having at least one parameter that is a type parameter. If T is a type parameter of a containing class, List<T> is an open constructed type.
Instantiation	Compiler generates real instruction code for methods defined in parameterized types and real virtual method table for closed constructed type.

See Also

Overview of Generics (see page 596)

Declaration of Generics (see page 598)

Overloads and Type Compatibility (Parameterized Types) (see page 602)

Constraints (Parameterized Types) (see page 603)

Class Variable in Parameterized Types (see page 605)

Changes in Standard Functions and Grammar (Parameterized Types) (see page 606)

3.1.3.4.3 Declaring Generics

The declaration of a generic is similar to the declaration of a regular class, record, or interface type. The difference is that a list of one or more type parameters placed between angle brackets (< and >) follows the type identifier in the declaration of a generic.

Type parameters can be used as a typical type identifier inside the container type declaration and method body.

For example:

```
type
  TPair<Tkey,TValue> = class   // TKey and TValue are type parameters
    FKey: TKey;
    FValue: TValue;
    function GetValue: TValue;
  end;
```

```
 function TPair<TKey,TValue>.GetValue: TValue;
begin
  Result := FValue;
end;
```

Type Argument

Generic types are instantiated by providing type arguments. In Delphi for .NET, you can use any type as a type argument except for the following: a static array, a short string, or a record type that (recursively) contains a field of one or more of these two types.

```
type
   TFoo<T> = class
     FData: T;
   end;
var
   F: TFoo<Integer>; // 'Integer' is sthe type argument of TFoo<T>
begin
   ,,,
end.
```

Nested Types

A nested type within a parameterized type is itself also a parameterized type.

```
type
  TFoo<T> = class
    type
      TBar = class
        X: Integer;
        // ...
      end;
    // ...  TBaz = class
    type
      TQux<T> = class
        X: Integer;
        // ...
      end;
    // ...
  end;
```

To access the TBar nested type, you must specify a construction of the TFoo type first:

```
var
  N: TFoo<Double>.TBar;
```

A parameterized type can also be declared within a regular class as a nested type:

```
type
  TOuter = class
    type
      TData<T> = class
        FFoo1: TFoo<Integer>;         // declared with closed constructed type        FFoo2:
TFoo<T>;               // declared with open constructed type
        FFooBar1: TFoo<Integer>.TBar; // declared with closed constructed type
        FFooBar2: TFoo<T>.TBar;       // declared with open constructed type
       FBazQux1: TBaz.TQux<Integer>; // declared with closed constructed type
        FBazQux2: TBaz.TQux<T>;       // declared with open constructed type
        ...
      end;
    var      FIntegerData: TData<Integer>;
      FStringData: TData<String>;
  end;
```

Base Types

The base type of a parameterized class or interface type might be an actual type or a constructed type. The base type might not be a type parameter alone.

```
type
  TFoo1<T> = class(TBar)            // Actual type
  end;

  TFoo2<T> =  class(TBar2<T>)       // Open constructed type
  end;
  TFoo3<T> = class(TBar3<Integer>)  // Closed constructed type
  end;
```

If TFoo2<String> is instantiated, an ancestor class becomes TBar2<String>, and TBar2<String> is automatically instantiated.

Class, Interface, and Record Types

Class, interface, and record types can be declared with type parameters.

For example:

```
type
   TRecord<T> = record
      FData: T;
   end;

type
   IAncestor<T> = interface
      function GetRecord: TRecrod<T>;
   end;
   IFoo<T> = interface(IAncestor<T>)
      procedure AMethod(Param: T);
end;

type
   TFoo<T> = class(TObject, IFoo<T>)
      FField: TRecord<T>;
      procedure AMethod(Param: T);
      function GetRecord: TRecord<T>;
   end;
```

Procedural Types

The procedure type and the method pointer can be declared with type parameters. Parameter types and result types can also use type parameters.

For example:

```
type
  TMyProc<T> = procedure(Param: T);
  TMyProc2<Y> = procedure(Param1, Param2: Y) of object;
type
  TFoo = class
    procedure Test;    procedure MyProc(X, Y: Integer);
  end;

procedure Sample(Param: Integer);
begin
  WriteLn(Param);
end;

procedure TFoo.MyProc(X, Y: Integer);
begin
  WriteLn('X:', X, ', Y:', Y);
end;
procedure TFoo.Test;
var
  X: TMyProc<Integer>;
  Y: TMyProc2<Integer>;
begin
  X := Sample;
  X(10);
  Y := MyProc;
  Y(20, 30);
end;

var
  F: TFoo;
begin
  F := TFoo.Create;
  F.Test;
  F.Free;
end.

var
   F: TFoo;
begin
   F := TFoo.Create;
   F.Test;
   F.Free;
end.
```

Parameterized Methods

Methods can be declared with type parameters. Parameter types and result types can use type parameters Parameterized methods are similar to overloaded methods.

There are two ways to instantiate a method:

- Explicitly specifying type argument
- Automatically inferring from the argument type

For example:

```
type
  TMyProc2<Y> = procedure(Param1, Param2: Y) of object;
  TFoo = class
    procedure Test;
    procedure MyProc2<T>(X, Y: T);
  end;

procedure TFoo.MyProc2<T>(X, Y: T);
begin
  Write('MyProc2<T>');
  {$IFDEF CIL}
  Write(X.ToString);
  Write(', ');
  WriteLn(Y.ToString);
  {$ENDIF}
  WR
end;

procedure TFoo.Test;
var
  P: TMyProc2<Integer>;
begin
  MyProc2<String>('Hello', 'World');
  MyProc2('Hello', 'World');
  MyProc2<Integer>(10, 20);
  MyProc2(10, 20);
  P := MyProc2<Integer>;
  P(40, 50);
end;

var
  F: TFoo;
begin
  F := TFoo.Create;
  F.Test;
  F.Free;
end.
```

Scope of Type Parameters

The scope of a type parameter covers the type declaration and the bodies of all its members, but does not include descendent types.

For example:

```
type
   TFoo<T> = class
     X: T;
   end;

   TBar<S> = class(TFoo<S>)
     Y: T;  // error!  unknown identifier "T"
   end;

var
  F: TFoo<Integer>;
begin
  F.T  // error! unknown identifier "T"
end.
```

See Also

Overview of Generics (Parameterized Types) (↗ see page 596)

Terminology of Generics (↗ see page 597)

Constraints in Generics (Parameterized Types) (↗ see page 603)

Overloads and Type Compatibility (Parameterized Types) (↗ see page 602)

Class Variable in Parameterized Types (↗ see page 605)

Changes in Standard Functions and Grammar (Parameterized Types) (↗ see page 606)

3.1.3.4.4 Overloads and Type Compatibility in Generics

Overloads

Generic methods can participate in overloading alongside non-generic methods by using the 'overload' directive. If overload selection between a generic method and a non-generic method would otherwise be ambiguous, the compiler selects the non-generic method.

For example:

```
type
  TFoo = class
    procedure Proc<T>(A: T);
    procedure Proc(A: String);
    procedure Test;
  end;

procedure TFoo.Test;
begin
  Proc('Hello'); // calls Proc(A: String);
  Proc<String>('Hello'); // calls Proc<T>(A: T);
end;
```

Type Compatibility

Two non-instantiated parameterized types are considered assignment compatible only if they are identical or are aliases to a common type.

Two instantiated parameterized types are considered assignment compatible if the base types are identical (or are aliases to a common type) and the type arguments are identical.

See Also

Overview of Generics (Parameterized Types) (see page 596)

Terminology of Generics (see page 597)

Declaring Parameterized Types (see page 598)

Constraints in Generics (Parameterized Types) (see page 603)

Class Variable in Parameterized Types (see page 605)

Changes in Standard Functions and Grammar (Parameterized Types) (see page 606)

3.1.3.4.5 Constraints in Generics

Constraints can be associated with a type parameter of a generic. Constraints declare items that must be supported by any concrete type passed to that parameter in a construction of the generic type.

Specifying Parameterized Types with Constraints

Constraint items include:

- Zero, one, or multiple interface types
- Zero or one class type
- The reserved word "constructor", "class", or "record"

You can specify both "constructor" and "class" for a constraint. However, "record" cannot be combined with other reserved words. Multiple constraints act as an additive union ("AND" logic)

The examples given here show only class types, although constraints apply to all forms of parameterized types

Declaring Constraints

Constraints are declared in a fashion that resembles type declarations in regular parameter lists:

```
type
  TFoo<T: ISerializable> = class
    FField: T;
  end;
```

In the example declaration given here, the 'T' type parameter indicates that it must support the ISerializable interface. In a type construction like TFoo<TMyClass>, the compiler checks at compile time to ensure that TMyClass actually implements ISerializable.

Multiple Type Parameters

When you specify constraints, you separate multiple type parameters by semicolons, as you do with a parameter list declaration:

```
type
   TFoo<T: ISerializable; V: IComparable>
```

Like parameter declarations, multiple type parameters can be grouped together in a comma list to bind to the same constraints:

```
type
   TFoo<S, U: ISerializable> ...
```

In the example above, S and U are both bound to the ISerializable constraint.

Multiple Constraints

Multiple constraints can be applied to a single type parameters as a comma list following the colon:

```
type
   TFoo<T: ISerializable, ICloneable; V: IComparable> ...
```

Constrained type parameters can be mixed with “free” type parameters. For example, all the following are valid:

```
type
   TFoo<T; C: IComparable> ...
   TBar<T, V> ...
   TTest<S: ISerializable; V> ...
   // T and V are free, but C and S are constrained
```

Types of Constraints

Interface Type Constraints

A type parameter constraint may contain zero, one, or a comma separated list of multiple interface types.

A type parameter constrained by an interface type means that the compiler will verify at compile time that a concrete type passed as an argument to a type construction implements the specified interface type(s).

For example:

```
type
  TFoo<T: ICloneable> ...

  TTest1 = class(TObject, ICloneable)
     ...
  end;

  TError = class
  end;

var
  X: TFoo<TTest1>;  //  TTest1 is checked for ICloneable support here
                    //  at compile time
  Y: TFoo<TError>;  //  exp: syntax error here - TError does not support
                    //  ICloneable
```

Class Type Constraints

A type parameter may be constrained by zero or one class type. As with interface type constraints, this declaration means that the compiler will require any concrete type passed as an argument to the constrained type param to be assignment compatible with the constraint class.

Compatibility of class types follows the normal rules of OOP type compatibilty - descendent types can be passed where their ancestor types are required.

Constructor Constraints

A type parameter may be constrained by zero or one instance of the reserved word "constructor". This means that the actual argument type must be a class that defines a default constructor (a public parameterless constructor), so that methods within the generic type may construct instances of the argument type using the argument type's default constructor, without knowing anything about the argument type itself (no minimum base type requirements).

In a constraint declaration, you can mix “constructor” in any order with interface or class type constraints.

Class Constraint

A type parameter may be constrained by zero or one instance of the reserved word "class". This means that the actual type must be a reference type, that is, a class or interface type.

Record Constraint

A type parameter may be constrained by zero or one instance of the reserved word "record". This means that the actual type must be a value type (not a reference type). A “record” constraint cannot be combined with a “class” or “constructor” constraint.

Type Inferencing

When using a field or variable of a constrained type parameter, it is not necessary in many cases to typecast in order to treat the field or variable as one of the constrained types. The compiler can infer which type you're referring to by looking at the method name and by performing a variation of overload resolution over the union of the methods sharing the same name across all the constraints on that type.

For example:

```
type
  TFoo<T: ISerializable, ICloneable> = class
    FData: T;
    procedure Test;
  end;

procedure TFoo<T>.Test;
begin
  FData.Clone;
end;
```

The compiler looks for "Clone" methods in ISerializable and ICloneable, since FData is of type T, which is guaranteed to support both those interfaces. If both interfaces implement "Clone" with the same parameter list, the compiler issues an ambiguous method call error and require you to typecast to one or the other interface to disambiguate the context.

See Also

Overview of Generics (Parameterized Types) (see page 596)

Terminology of Generics (see page 597)

Declaring Generics (Parameterized Types) (see page 598)

Overloads and Type Compatibility (Parameterized Types) (see page 602)

Class Variable in Parameterized Types (see page 605)

Changes in Standard Functions and Grammar (Parameterized Types) (see page 606)

3.1.3.4.6 Class Variable in Generics

The class variable defined in a generic type is instantiated in each instantiated type identified by the type parameters.

The following code shows that TFoo<Integer>.FCount and TFoo<String>.FCount are instantiated only once, and these are two different variables.

```
{$APPTYPE CONSOLE}
type
  TFoo<T> = class
    class var FCount: Integer;
    constructor Create;
  end;
  constructor TFoo<T>.Create;
begin
  inherited Create;
  Inc(FCount);
end;

procedure Test;
```

```
  FI: TFoo<Integer>;
begin
  FI := TFoo<Integer>.Create;
  FI.Free;
end;

var
  FI: TFoo<Integer>;
  FS: TFoo<String>;

begin
  FI := TFoo<Integer>.Create;
  FI.Free;
  FS := TFoo<String>.Create;
  FS.Free;
  Test;
  WriteLn(TFoo<Integer>.FCount); // outputs 2
  WriteLn(TFoo<String>.FCount);  // outputs 1
end;
```

See Also

Overview of Generics (Parameterized Types) (see page 596)

Terminology of Generics (see page 597)

Declaring Parameterized Types (see page 598)

Constraints in Generics (Parameterized Types) (see page 603)

Overloads and Type Compatibility (Parameterized Types) (see page 602)

Changes in Standard Functions and Grammar (Parameterized Types) (see page 606)

3.1.3.4.7 Changes in Standard Functions and Grammar

Here is a list of standard function changes to support parameterized types. Example forms:

```
    Instantiated type      : TFoo<Integer,String>
    Open constructed type  : TFoo<Integer,T>
    Parameterized type     : TFoo<,>

procedure Initialize(var X);                                        [.Net]

    Instantiated type      : allowed
    Open constructed type  : allowed     => Not yet implemented.
    Parameterized type     : NOT allowed
function High(X:TypeId): Integer|Int64|UInt64;
function Low(X:TypeId): Integer|Int64|UInt64;
    Instantiated type      : allowed
    Open constructed type  : allowed     => Not yet implemented.
    Parameterized type     : NOT allowed

function Default(X:TYPE_ID):  valueOfTypeId>;
    Instantiated type      : allowed
    Open constructed type  : allowed     => Not yet implemented.
    Parameterized type     : NOT allowed

function New;
// a := New(array[2,3] of Integer);                                 [.Net]
// a := New(array[,], (const array init expr));                     [.Net]
// a := New(dynArrayTypeId, dim1 [, dim2...dimN]);                  [.Net]
    Instantiated type      : allowed
    Open constructed type  : allowed     => Not yet implemented.
    Parameterized type     : NOT allowed
```

```
function SizeOf(TYPE_ID): PosInt;
    Instantiated type      : allowed
    Open constructed type  : allowed
    Parameterized type : NOT allowed

function TypeInfo;
function TypeHandle;
function TypeId;
// function TypeHandle(Identifier): System.RuntimeTypeHandle; [.Net]
// function TypeInfo(Identifier): System.Type;                [.Net]
// function TypeOf(Identifier): System.Type;                  [.Net]
    Instantiated type      : allowed
    Open constructed type  : allowed     => Not yet implemented
    Parameterized type : NOT allowed
```

Delphi Language Grammar Changes

These changes are in support of generics or parameterized types.

```
{ Type Declarations }

TypeDeclaration -> [ CAttrs ] Ident '=' Type
                -> [ CAttrs ] Ident '=' RecordTypeDecl
                -> [ CAttrs ] Ident '=' ClassTypeDecl
                -> [ CAttrs ] Ident '=' InterfaceTypeDecl
                -> [ CAttrs ] Ident '=' ClassHelperTypeDecl
                -> [ CAttrs ] Ident '=' RecordHelperTypeDecl
                 -> [ CAttrs ] Ident '=' TYPE TypeId
                -> [ CAttrs ] Ident '=' TYPE ClassTypeId            {.Net only}
{NEW}           -> [ CAttrs ] Ident TypeParams '=' RecordTypeDecl
{NEW}           -> [ CAttrs ] Ident TypeParams '=' ClassTypeDecl
{NEW}           -> [ CAttrs ] Ident TypeParams '=' InterfaceTypeDecl
{NEW}           -> [ CAttrs ] Ident TypeParams '=' Type

{NEW} TypeParams -> '<' TypeParamDeclList '>'

{NEW} TypeParamDeclList -> TypeParamDecl/';'...

{NEW} TypeParamDecl -> TypeParamList [ ':' ConstraintList ]

{NEW} TypeParamList -> ( [ CAttrs ] [ '+' | '-' [ CAttrs ] ] Ident )/','...

{NEW} ConstraintList -> Constraint/','...

{NEW} Constraint -> CONSTRUCTOR
{NEW}            -> RECORD
{NEW}            -> CLASS
{NEW}            -> TypeId

MethodResolutionClause -> FUNCTION InterfaceIdent '.'
{OLD}                     Ident               '=' Ident             ';'
{NEW}                     Ident [ TypeArgs ]  '=' Ident [ TypeArgs ] ';'
                       -> PROCEDURE InterfaceIdent '.'
{OLD}                     Ident               '=' Ident             ';'
{NEW}                     Ident [ TypeArgs ]  '=' Ident [ TypeArgs ] ';'

FunctionHeading -> [ CLASS ] FUNCTION Ident
{NEW}              [ FormalTypeParamList ]
                   [ FormalParameterList ] ':' TypeIdStringFile

ProcedureHeading -> [ CALSS ] PROCEDURE Ident
{NEW}               [ FormalTypeParamList ]
                    [ FormalParameterList ]

ClassOperatorHeading -> CLASS OPERATOR OperatorIdent
{NEW}                   [ FormalTypeParamList ]
                        FormalParameterList : TypeIdStringFile
```

```
ConstructorHeading -> CONSTRUCTOR Ident
{NEW}                 [ FormalTypeParamList ]
                      [ FormalParameterList ]

RecordConstructorHeading -> CONSTRUCTOR Ident
{NEW}                       [ FormalTypeParamList ]
                            FormalParameterList

DestructorHeading -> DESTRUCTOR Ident
{NEW}                [ FormalTypeParamList ]
                     [ FormalParameterList ]

MethodBodyHeading -> [ CLASS ] FUNCTION NSTypeId '.' Ident
{NEW}                [ FormalTypeParamList ]
                     [ FormalParameterList ] ':' TypeIdStringFile
                  -> [ CLASS ] PROCEDURE NSTypeId '.' Ident
{NEW}                [ FormalTypeParamList ]
                     [ FormalParameterList ]

ProcedureTypeHeading -> PROCEDURE
{NEW}                   [ FormalTypeParamList ]
                        [ FormalParameterList ]

FunctionTypeHeading -> FUNCTION
{NEW}                  [ FormalTypeParamList ]
                       [ FormalParameterList ] ':' TypeIdStringFile

FormalTypeParamList -> '<' TypeParamDeclList >'


{ Types }

Type -> TypeId
     -> SimpleType
     -> StructualType
     -> PointerType
     -> StringType
     -> ProcedureType

     -> ClassRefType
     -> TypeRefType
{NEW}-> ClassTypeId TypeArgs
{NEW}-> RecordTypeId TypeArgs
{NEW}-> InterfaceIdent TypeArgs

{NEW} TypeArgs -> '<' ( TypeId | STRING )/','... '>'


{ Attributes }

CAttrExpr -> ConstExpr
          -> TYPEOF '(' TypeId ')'
{NEW}     -> TYPEOF '(' TypeId '<' [ ','... ] '>' ')'
{NEW}     -> TYPEOF '(' TypeId '<' TypeId/','... '>' ')'
```

See Also

Overview of Generics (Parameterized Types) (↗ see page 596)

Terminology of Generics (↗ see page 597)

Declaring Generics (Parameterized Types) (↗ see page 598)

Overloads and Type Compatibility (Parameterized Types) (↗ see page 602)

Constraints (Parameterized Types) (↗ see page 603)

Class Variable in Parameterized Types (↗ see page 605)

3.1.3.5 Inline Assembly Code (Win32 Only)

This section describes the use of the inline assembler on the Win32 platform.

Topics

Name	Description
Using Inline Assembly Code (Win32 Only) (↗ see page 610)	The built-in assembler allows you to write assembly code within Delphi programs. The inline assembler is available only on the Win32 Delphi compiler. It has the following features: • Allows for inline assembly. • Supports all instructions found in the Intel Pentium 4, Intel MMX extensions, Streaming SIMD Extensions (SSE), and the AMD Athlon (including 3D Now!). • Provides no macro support, but allows for pure assembly function procedures. • Permits the use of Delphi identifiers, such as constants, types, and variables in assembly statements. As an alternative to the built-in assembler, you can link to object files that contain external procedures and functions.... more (↗ see page 610)
Understanding Assembler Syntax (Win32 Only) (↗ see page 610)	The inline assembler is available only on the Win32 Delphi compiler. The following material describes the elements of the assembler syntax necessary for proper use. • Assembler Statement Syntax • Labels • Instruction Opcodes • Assembly Directives • Operands
Assembly Expressions (Win32 Only) (↗ see page 616)	The built-in assembler evaluates all expressions as 32-bit integer values. It doesn't support floating-point and string values, except string constants. The inline assembler is available only on the Win32 Delphi compiler. Expressions are built from expression elements and operators, and each expression has an associated expression class and expression type. This topic covers the following material: • Differences between Delphi and Assembler Expressions • Expression Elements • Expression Classes • Expression Types • Expression Operators
Assembly Procedures and Functions (Win32 Only) (↗ see page 623)	You can write complete procedures and functions using inline assembly language code, without including a `begin...end` statement. This topic covers these issues: • Compiler Optimizations. • Function Results. The inline assembler is available only on the Win32 Delphi compiler.

3.1.3.5.1 Using Inline Assembly Code (Win32 Only)

The built-in assembler allows you to write assembly code within Delphi programs. The inline assembler is available only on the Win32 Delphi compiler. It has the following features:

- Allows for inline assembly.
- Supports all instructions found in the Intel Pentium 4, Intel MMX extensions, Streaming SIMD Extensions (SSE), and the AMD Athlon (including 3D Now!).
- Provides no macro support, but allows for pure assembly function procedures.
- Permits the use of Delphi identifiers, such as constants, types, and variables in assembly statements.

As an alternative to the built-in assembler, you can link to object files that contain external procedures and functions. See the topic on External declarations for more information. If you have external assembly code that you want to use in your applications, you should consider rewriting it in the Delphi language or minimally reimplement it using the inline assembler.

Using the asm Statement

The built-in assembler is accessed through **asm** statements, which have the form

```
asm statementList end
```

where *statementList* is a sequence of assembly statements separated by semicolons, end-of-line characters, or Delphi comments.

Comments in an **asm** statement must be in Delphi style. A semicolon does not indicate that the rest of the line is a comment.

The reserved word **inline** and the directive **assembler** are maintained for backward compatibility only. They have no effect on the compiler.

Using Registers

In general, the rules of register use in an **asm** statement are the same as those of an **external** procedure or function. An **asm** statement must preserve the EDI, ESI, ESP, EBP, and EBX registers, but can freely modify the EAX, ECX, and EDX registers. On entry to an **asm** statement, EBP points to the current stack frame and ESP points to the top of the stack. Except for ESP and EBP, an **asm** statement can assume nothing about register contents on entry to the statement.

See Also

Understanding Assembler Syntax (↗ see page 610)

Assembly Expressions (↗ see page 616)

Assembly Procedures and Functions (↗ see page 623)

3.1.3.5.2 Understanding Assembler Syntax (Win32 Only)

The inline assembler is available only on the Win32 Delphi compiler. The following material describes the elements of the assembler syntax necessary for proper use.

- Assembler Statement Syntax
- Labels
- Instruction Opcodes
- Assembly Directives
- Operands

Assembler Statement Syntax

This syntax of an assembly statement is

```
Label: Prefix Opcode Operand1, Operand2
```

where *Label* is a label, *Prefix* is an assembly prefix opcode (operation code), *Opcode* is an assembly instruction opcode or directive, and *Operand* is an assembly expression. Label and Prefix are optional. Some opcodes take only one operand, and some take none.

Comments are allowed between assembly statements, but not within them. For example,

```
     MOV AX,1 {Initial value}           { OK }
     MOV CX,100 {Count}                                                         { OK }

                    MOV {Initial value} AX,1;                               { Error! }
     MOV CX, {Count} 100                                                    { Error! }
```

Labels

Labels are used in built-in assembly statements as they are in the Delphi language by writing the label and a colon before a statement. There is no limit to a label's length. As in Delphi, labels must be declared in a **label** declaration part in the block containing the **asm** statement. The one exception to this rule is local labels.

Local labels are labels that start with an at-sign (@). They consist of an at-sign followed by one or more letters, digits, underscores, or at-signs. Use of local labels is restricted to **asm** statements, and the scope of a local label extends from the **asm** reserved word to the end of the **asm** statement that contains it. A local label doesn't have to be declared.

Instruction Opcodes

The built-in assembler supports all of the Intel-documented opcodes for general application use. Note that operating system privileged instructions may not be supported. Specifically, the following families of instructions are supported:

- Pentium family
- Pentium Pro and Pentium II
- Pentium III
- Pentium 4

In addition, the built-in assembler supports the following instruction sets

- AMD 3DNow! (from the AMD K6 onwards)
- AMD Enhanced 3DNow! (from the AMD Athlon onwards)

For a complete description of each instruction, refer to your microprocessor documentation.

RET instruction sizing

The RET instruction opcode always generates a near return.

Automatic jump sizing

Unless otherwise directed, the built-in assembler optimizes jump instructions by automatically selecting the shortest, and therefore most efficient, form of a jump instruction. This automatic jump sizing applies to the unconditional jump instruction (JMP), and to all conditional jump instructions when the target is a label (not a procedure or function).

For an unconditional jump instruction (JMP), the built-in assembler generates a short jump (one-byte opcode followed by a one-byte displacement) if the distance to the target label is 128 to 127 bytes. Otherwise it generates a near jump (one-byte opcode followed by a two-byte displacement).

For a conditional jump instruction, a short jump (one-byte opcode followed by a one-byte displacement) is generated if the distance to the target label is 128 to 127 bytes. Otherwise, the built-in assembler generates a short jump with the inverse condition, which jumps over a near jump to the target label (five bytes in total). For example, the assembly statement

```
JC    Stop
```

where **Stop** isn't within reach of a short jump, is converted to a machine code sequence that corresponds to this:

```
     JNC    Skip
     JMP    Stop
     Skip:
```

Jumps to the entry points of procedures and functions are always near.

Assembly Directives

The built-in assembler supports three assembly define directives: DB (define byte), DW (define word), and DD (define double word). Each generates data corresponding to the comma-separated operands that follow the directive.

Directive	Description
DB	Define byte: generates a sequence of bytes. Each operand can be a constant expression with a value between 128 and 255, or a character string of any length. Constant expressions generate one byte of code, and strings generate a sequence of bytes with values corresponding to the ASCII code of each character.
DW	Define word: generates a sequence of words. Each operand can be a constant expression with a value between 32,768 and 65,535, or an address expression. For an address expression, the built-in assembler generates a near pointer, a word that contains the offset part of the address.
DD	Define double word: generates a sequence of double words. Each operand can be a constant expression with a value between 2,147,483,648 and 4,294,967,295, or an address expression. For an address expression, the built-in assembler generates a far pointer, a word that contains the offset part of the address, followed by a word that contains the segment part of the address.
DQ	Define quad word: defines a quad word for Int64 values.

The data generated by the DB, DW, and DD directives is always stored in the code segment, just like the code generated by other built-in assembly statements. To generate uninitialized or initialized data in the data segment, you should use Delphi **var** or **const** declarations.

Some examples of DB, DW, and DD directives follow.

```
 asm
     DB
FFH

{ One byte }
     DB
0,99                                                                                 {
Two bytes }
     DB
'A'

{ Ord('A') }
     DB         'Hello world...',0DH,0AH        { String followed by CR/LF }
     DB         12,'string'                                                      {
Delphi style string }
     DW
0FFFFH                                                                       { One word
}
     DW
0,9999                                                                           { Two
words }
     DW
'A'

{ Same as DB  'A',0 }
     DW
'BA'                                                                                 {
Same as DB 'A','B' }
     DW
MyVar                                                                            {
Offset of MyVar }
```

```
     DW
MyProc                                                                               { Offset
of MyProc }
     DD        0FFFFFFFFH                                                                { One
double-word }
     DD        0,999999999                                                               { Two
double-words }
     DD
'A'

{ Same as DB 'A',0,0,0 }
     DD
'DCBA'                                                                               { Same as
DB 'A','B','C','D' }
     DD
MyVar                                                                                    {
Pointer to MyVar }
     DD
MyProc                                                                               { Pointer
to MyProc }
 end;
```

When an identifier precedes a DB, DW , or DD directive, it causes the declaration of a byte-, word-, or double-word-sized variable at the location of the directive. For example, the assembler allows the following:

```
     ByteVar        DB       ?
     WordVar        DW      ?
     IntVar         DD      ?
                    .
                    .
                    .
                MOV         AL,ByteVar
                MOV         BX,WordVar
                MOV         ECX,IntVar
```

The built-in assembler doesn't support such variable declarations. The only kind of symbol that can be defined in an inline assembly statement is a label. All variables must be declared using Delphi syntax; the preceding construction can be replaced by

```
var
     ByteVar: Byte;
     WordVar: Word;
     IntVar: Integer;
                    .
                    .
                    .

    asm
     MOV AL,ByteVar
     MOV BX,WordVar
     MOV ECX,IntVar
    end;
```

SMALL and LARGE can be used determine the width of a displacement:

```
MOV EAX, [LARGE $1234]
```

This instruction generates a 'normal' move with a 32-bit displacement ($00001234).

```
MOV EAX, [SMALL $1234]
```

The second instruction will generate a move with an address size override prefix and a 16-bit displacement ($1234).

SMALL can be used to save space. The following example generates an address size override and a 2-byte address (in total three bytes)

```
 MOV EAX, [SMALL 123]
```

as opposed to

```
 MOV EAX, [123]
```

which will generate no address size override and a 4-byte address (in total four bytes).

Two additional directives allow assembly code to access dynamic and virtual methods: VMTOFFSET and DMTINDEX.

VMTOFFSET retrieves the offset in bytes of the virtual method pointer table entry of the virtual method argument from the beginning of the virtual method table (VMT). This directive needs a fully specified class name with a method name as a parameter (for example, TExample.VirtualMethod), or an interface name and an interface method name.

DMTINDEX retrieves the dynamic method table index of the passed dynamic method. This directive also needs a fully specified class name with a method name as a parameter, for example, TExample.DynamicMethod. To invoke the dynamic method, call System.@CallDynaInst with the (E)SI register containing the value obtained from DMTINDEX.

Note: Methods with the *message* directive are implemented as dynamic methods and can also be called using the DMTINDEX technique. For example:

```
  TMyClass = class
     procedure x; message MYMESSAGE;
  end;
```

The following example uses both DMTINDEX and VMTOFFSET to access dynamic and virtual methods:

```
program Project2;
   type
    TExample = class
     procedure DynamicMethod; dynamic;
        procedure VirtualMethod; virtual;
    end;

                procedure TExample.DynamicMethod;
    begin

             end;

                procedure TExample.VirtualMethod;
    begin

             end;

                procedure CallDynamicMethod(e: TExample);
     asm
         // Save ESI register
        PUSH    ESI

                        // Instance pointer needs to be in EAX
        MOV     EAX, e

                        // DMT entry index needs to be in (E)SI
        MOV     ESI, DMTINDEX TExample.DynamicMethod

                        // Now call the method
        CALL    System.@CallDynaInst

                        // Restore ESI register
        POP ESI

                    end;

                    procedure CallVirtualMethod(e: TExample);
     asm
            // Instance pointer needs to be in EAX
            MOV     EAX, e
```

```
                            // Retrieve VMT table entry
            MOV     EDX, [EAX]

                            // Now call the method at offset VMTOFFSET
            CALL    DWORD PTR [EDX + VMTOFFSET TExample.VirtualMethod]

                        end;

                    var
        e: TExample;
     begin
        e := TExample.Create;
            try
                CallDynamicMethod(e);
                CallVirtualMethod(e);
            finally
                e.Free;
     end;
    end.
```

Operands

Inline assembler operands are expressions that consist of constants, registers, symbols, and operators.

Within operands, the following reserved words have predefined meanings:

Built-in assembler reserved words

AH	CL	DX	ESP	mm4	SHL	WORD
AL	CS	EAX	FS	mm5	SHR	xmm0
AND	CX	EBP	GS	mm6	SI	xmm1
AX	DH	EBX	HIGH	mm7	SMALL	xmm2
BH	DI	ECX	LARGE	MOD	SP	xmm3
BL	DL	EDI	LOW	NOT	SS	xmm4
BP	CL	EDX	mm0	OFFSET	ST	xmm5
BX	DMTINDEX	EIP	mm1	OR	TBYTE	xmm6
BYTE	DS	ES	mm2	PTR	TYPE	xmm7
CH	DWORD	ESI	mm3	QWORD	VMTOFFSET	XOR

Reserved words always take precedence over user-defined identifiers. For example,

```
var
  Ch: Char;
        .
        .
        .
asm
  MOV   CH, 1
end;
```

loads 1 into the CH register, not into the *Ch* variable. To access a user-defined symbol with the same name as a reserved word, you must use the ampersand (**&**) override operator:

```
MOV&Ch, 1
```

It is best to avoid user-defined identifiers with the same names as built-in assembler reserved words.

See Also

Using Inline Assembly Code (↗ see page 610)

Assembly Expressions (see page 616)

Assembly Procedures and Functions (see page 623)

3.1.3.5.3 Assembly Expressions (Win32 Only)

The built-in assembler evaluates all expressions as 32-bit integer values. It doesn't support floating-point and string values, except string constants. The inline assembler is available only on the Win32 Delphi compiler.

Expressions are built from expression elements and operators, and each expression has an associated expression class and expression type. This topic covers the following material:

- Differences between Delphi and Assembler Expressions
- Expression Elements
- Expression Classes
- Expression Types
- Expression Operators

Differences between Delphi and Assembler Expressions

The most important difference between Delphi expressions and built-in assembler expressions is that assembler expressions must resolve to a constant value. In other words, it must resolve to a value that can be computed at compile time. For example, given the declarations

```
const
 X = 10;
 Y = 20;
var
 Z: Integer;
```

the following is a valid statement.

```
asm
 MOV        Z,X+Y
end;
```

Because both X and Y are constants, the expression `X + Y` is a convenient way of writing the constant 30, and the resulting instruction simply moves of the value 30 into the variable Z. But if X and Y are variables

```
var
 X, Y: Integer;
```

the built-in assembler cannot compute the value of `X + Y` at compile time. In this case, to move the sum of X and Y into Z you would use

```
asm
 MOV            EAX,X
 ADD            EAX,Y
 MOV            Z,EAX
end;
```

In a Delphi expression, a variable reference denotes the *contents* of the variable. But in an assembler expression, a variable reference denotes the *address* of the variable. In Delphi the expression `X + 4` (where X is a variable) means the contents of X plus 4, while to the built-in assembler it means the contents of the word at the address four bytes higher than the address of X. So, even though you are allowed to write

```
asm
 MOV            EAX,X+4
end;
```

this code doesn't load the value of X plus 4 into AX; instead, it loads the value of a word stored four bytes beyond X. The correct way to add 4 to the contents of X is

```
asm
 MOV            EAX,X
    ADD         EAX,4
end;
```

Expression Elements

The elements of an expression are constants, registers, and symbols.

Numeric Constants

Numeric constants must be integers, and their values must be between 2,147,483,648 and 4,294,967,295.

By default, numeric constants use decimal notation, but the built-in assembler also supports binary, octal, and hexadecimal. Binary notation is selected by writing a B after the number, octal notation by writing an O after the number, and hexadecimal notation by writing an H after the number or a **$** before the number.

Numeric constants must start with one of the digits 0 through 9 or the **$** character. When you write a hexadecimal constant using the H suffix, an extra zero is required in front of the number if the first significant digit is one of the digits A through F. For example, 0BAD4H and $BAD4 are hexadecimal constants, but BAD4H is an identifier because it starts with a letter.

String Constants

String constants must be enclosed in single or double quotation marks. Two consecutive quotation marks of the same type as the enclosing quotation marks count as only one character. Here are some examples of string constants:

```
 'Z'
 'Delphi'
 'Linux'
 "That's all folks"
 '"That''s all folks," he said.'
 '100'
 '"'
 "'"
```

String constants of any length are allowed in DB directives, and cause allocation of a sequence of bytes containing the ASCII values of the characters in the string. In all other cases, a string constant can be no longer than four characters and denotes a numeric value which can participate in an expression. The numeric value of a string constant is calculated as

```
Ord(Ch1) + Ord(Ch2) shl 8 + Ord(Ch3) shl 16 + Ord(Ch4) shl 24
```

where `Ch1` is the rightmost (last) character and `Ch4` is the leftmost (first) character. If the string is shorter than four characters, the leftmost characters are assumed to be zero. The following table shows string constants and their numeric values.

String examples and their values

String	Value
'a'	00000061H
'ba'	00006261H
'cba'	00636261H
'dcba'	64636261H
'a '	00006120H
' a'	20202061H
'a' * 2	000000E2H
'a'-'A'	00000020H
not 'a'	FFFFFF9EH

Registers

The following reserved symbols denote CPU registers in the inline assembler:

CPU registers

32-bit general purpose	EAX EBX ECX EDX	32-bit pointer or index	ESP EBP ESI EDI
16-bit general purpose	AX BX CX DX	16-bit pointer or index	SP BP SI DI
8-bit low registers	AL BL CL DL	16-bit segment registers	CS DS SS ES
		32-bit segment registers	FS GS
8-bit high registers	AH BH CH DH	Coprocessor register stack	ST

When an operand consists solely of a register name, it is called a register operand. All registers can be used as register operands, and some registers can be used in other contexts.

The base registers (BX and BP) and the index registers (SI and DI) can be written within square brackets to indicate indexing. Valid base/index register combinations are [BX], [BP], [SI], [DI], [BX+SI], [BX+DI], [BP+SI], and [BP+DI]. You can also index with all the 32-bit registersfor example, [EAX+ECX], [ESP], and [ESP+EAX+5].

The segment registers (ES, CS, SS, DS, FS, and GS) are supported, but segments are normally not useful in 32-bit applications.

The symbol ST denotes the topmost register on the 8087 floating-point register stack. Each of the eight floating-point registers can be referred to using ST(X), where X is a constant between 0 and 7 indicating the distance from the top of the register stack.

Symbols

The built-in assembler allows you to access almost all Delphi identifiers in assembly language expressions, including constants, types, variables, procedures, and functions. In addition, the built-in assembler implements the special symbol **@Result**, which corresponds to the *Result* variable within the body of a function. For example, the function

```
function Sum(X, Y: Integer): Integer;
begin
        Result := X + Y;
end;
```

could be written in assembly language as

```
function Sum(X, Y: Integer): Integer; stdcall;
begin
 asm
         MOV        EAX,X
            ADD     EAX,Y
            MOV     @Result,EAX
 end;
end;
```

The following symbols cannot be used in **asm** statements:

- Standard procedures and functions (for example, **WriteLn** and **Chr**).
- String, floating-point, and set constants (except when loading registers).
- Labels that aren't declared in the current block.
- The **@Result** symbol outside of functions.

The following table summarizes the kinds of symbol that can be used in **asm** statements.

Symbols recognized by the built-in assembler

Symbol	Value	Class	Type
Label	Address of label	Memory reference	Size of type
Constant	Value of constant	Immediate value	0
Type	0	Memory reference	Size of type
Field	Offset of field	Memory	Size of type
Variable	Address of variable or address of a pointer to the variable	Memory reference	Size of type
Procedure	Address of procedure	Memory reference	Size of type
Function	Address of function	Memory reference	Size of type
Unit	0	Immediate value	0
@Result	Result variable offset	Memory reference	Size of type

With optimizations disabled, local variables (variables declared in procedures and functions) are always allocated on the stack and accessed relative to EBP, and the value of a local variable symbol is its signed offset from EBP. The assembler automatically adds [EBP] in references to local variables. For example, given the declaration

```
var Count: Integer;
```

within a function or procedure, the instruction

```
MOV       EAX,Count
```

assembles into `MOV EAX,[EBP4]`.

The built-in assembler treats **var** parameters as a 32-bit pointers, and the size of a **var** parameter is always 4. The syntax for accessing a **var** parameter is different from that for accessing a value parameter. To access the contents of a **var** parameter, you must first load the 32-bit pointer and then access the location it points to. For example,

```
function Sum(var X, Y: Integer): Integer; stdcall;
    begin
        asm
                MOV             EAX,X
                MOV             EAX,[EAX]
                MOV             EDX,Y
                ADD             EAX,[EDX]
                MOV             @Result,EAX
     end;
    end;
```

Identifiers can be qualified within **asm** statements. For example, given the declarations

```
 type
     TPoint = record
            X, Y: Integer;
     end;
     TRect = record
            A, B: TPoint;
     end;
    var
     P: TPoint;
     R: TRect;
```

the following constructions can be used in an **asm** statement to access fields.

```
MOV         EAX,P.X
MOV         EDX,P.Y
MOV         ECX,R.A.X
MOV         EBX,R.B.Y
```

A type identifier can be used to construct variables on the fly. Each of the following instructions generates the same machine code, which loads the contents of [EDX] into EAX.

```
MOV         EAX,(TRect PTR [EDX]).B.X
MOV         EAX,TRect([EDX]).B.X
MOV         EAX,TRect[EDX].B.X
MOV         EAX,[EDX].TRect.B.X
```

Expression Classes

The built-in assembler divides expressions into three classes: registers, memory references, and immediate values.

An expression that consists solely of a register name is a register expression. Examples of register expressions are AX, CL, DI, and ES. Used as operands, register expressions direct the assembler to generate instructions that operate on the CPU registers.

Expressions that denote memory locations are memory references. Delphi's labels, variables, typed constants, procedures, and functions belong to this category.

Expressions that aren't registers and aren't associated with memory locations are immediate values. This group includes Delphi's untyped constants and type identifiers.

Immediate values and memory references cause different code to be generated when used as operands. For example,

```
const
 Start = 10;
var
 Count: Integer;
    .
    .
    .
asm
 MOV            EAX,Start                                       { MOV EAX,xxxx }
 MOV            EBX,Count                                       { MOV EBX,[xxxx] }
 MOV            ECX,[Start]                             { MOV ECX,[xxxx] }
 MOV            EDX,OFFSET Count            { MOV EDX,xxxx }
end;
```

Because Start is an immediate value, the first MOV is assembled into a move immediate instruction. The second MOV, however, is translated into a move memory instruction, as Count is a memory reference. In the third MOV, the brackets convert Start into a memory reference (in this case, the word at offset 10 in the data segment). In the fourth MOV, the OFFSET operator converts Count into an immediate value (the offset of Count in the data segment).

The brackets and OFFSET operator complement each other. The following **asm** statement produces identical machine code to the first two lines of the previous **asm** statement.

```
asm
 MOV            EAX,OFFSET [Start]
 MOV            EBX,[OFFSET Count]
end;
```

Memory references and immediate values are further classified as either relocatable or absolute. Relocation is the process by which the linker assigns absolute addresses to symbols. A relocatable expression denotes a value that requires relocation at link time, while an absolute expression denotes a value that requires no such relocation. Typically, expressions that refer to labels, variables, procedures, or functions are relocatable, since the final address of these symbols is unknown at compile time. Expressions that operate solely on constants are absolute.

The built-in assembler allows you to carry out any operation on an absolute value, but it restricts operations on relocatable values to addition and subtraction of constants.

Expression Types

Every built-in assembler expression has a type, or more correctly a size, because the assembler regards the type of an expression simply as the size of its memory location. For example, the type of an Integer variable is four, because it occupies 4 bytes. The built-in assembler performs type checking whenever possible, so in the instructions

```
var
 QuitFlag: Boolean;
 OutBufPtr: Word;
 .
    .
    .
asm
 MOV            AL,QuitFlag
 MOV            BX,OutBufPtr
end;
```

the assembler checks that the size of `QuitFlag` is one (a byte), and that the size of `OutBufPtr` is two (a word). The instruction

```
MOV           DL,OutBufPtr
```

produces an error because DL is a byte-sized register and `OutBufPtr` is a word. The type of a memory reference can be changed through a typecast; these are correct ways of writing the previous instruction:

```
MOV         DL,BYTE PTR OutBufPtr
MOV         DL,Byte(OutBufPtr)
MOV         DL,OutBufPtr.Byte
```

These MOV instructions all refer to the first (least significant) byte of the `OutBufPtr` variable.

In some cases, a memory reference is untyped. One example is an immediate value (Buffer) enclosed in square brackets:

```
procedure Example(var Buffer);
     asm
        MOV AL,     [Buffer]
        MOV CX,     [Buffer]
        MOV EDX, [Buffer]
     end;
```

The built-in assembler permits these instructions, because the expression [Buffer] has no type. [Buffer] means "the contents of the location indicated by Buffer," and the type can be determined from the first operand (byte for AL, word for CX, and double-word for EDX).

In cases where the type can't be determined from another operand, the built-in assembler requires an explicit typecast. For example,

```
INC     BYTE PTR [ECX]
IMUL    WORD PTR [EDX]
```

The following table summarizes the predefined type symbols that the built-in assembler provides in addition to any currently declared Delphi types.

Predefined type symbols

Symbol	Type
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10

Expression Operators

The built-in assembler provides a variety of operators. Precedence rules are different from that of the Delphi language; for example, in an **asm** statement, AND has lower precedence than the addition and subtraction operators. The following table lists the built-in assembler's expression operators in decreasing order of precedence.

Precedence of built-in assembler expression operators

Operators	Remarks	Precedence
&		highest
(...), **[...]**,**.**, **HIGH**, **LOW**		
+, **-**	unary **+** and **-**	
:		
OFFSET, **TYPE**, **PTR**, *****, **/**, **MOD**, **SHL**, **SHR**, **+**, **-**	binary **+** and **-**	
NOT, **AND**, **OR**, **XOR**		lowest

The following table defines the built-in assembler's expression operators.

Definitions of built-in assembler expression operators

Operator	Description
&	**Identifier override**. The identifier immediately following the ampersand is treated as a user-defined symbol, even if the spelling is the same as a built-in assembler reserved symbol.
(...)	**Subexpression**. Expressions within parentheses are evaluated completely prior to being treated as a single expression element. Another expression can precede the expression within the parentheses; the result in this case is the sum of the values of the two expressions, with the type of the first expression.
[...]	**Memory reference**. The expression within brackets is evaluated completely prior to being treated as a single expression element. Another expression can precede the expression within the brackets; the result in this case is the sum of the values of the two expressions, with the type of the first expression. The result is always a memory reference.
.	**Structure member selector**. The result is the sum of the expression before the period and the expression after the period, with the type of the expression after the period. Symbols belonging to the scope identified by the expression before the period can be accessed in the expression after the period.
HIGH	Returns the high-order 8 bits of the word-sized expression following the operator. The expression must be an absolute immediate value.
LOW	Returns the low-order 8 bits of the word-sized expression following the operator. The expression must be an absolute immediate value.
+	**Unary plus**. Returns the expression following the plus with no changes. The expression must be an absolute immediate value.
-	**Unary minus**. Returns the negated value of the expression following the minus. The expression must be an absolute immediate value.
+	**Addition**. The expressions can be immediate values or memory references, but only one of the expressions can be a relocatable value. If one of the expressions is a relocatable value, the result is also a relocatable value. If either of the expressions is a memory reference, the result is also a memory reference.
-	**Subtraction**. The first expression can have any class, but the second expression must be an absolute immediate value. The result has the same class as the first expression.
:	**Segment override**. Instructs the assembler that the expression after the colon belongs to the segment given by the segment register name (CS, DS, SS, FS, GS, or ES) before the colon. The result is a memory reference with the value of the expression after the colon. When a segment override is used in an instruction operand, the instruction is prefixed with an appropriate segment-override prefix instruction to ensure that the indicated segment is selected.
OFFSET	Returns the offset part (double word) of the expression following the operator. The result is an immediate value.
TYPE	Returns the type (size in bytes) of the expression following the operator. The type of an immediate value is 0.

PTR	**Typecast operator**. The result is a memory reference with the value of the expression following the operator and the type of the expression in front of the operator.
*****	**Multiplication**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
/	**Integer division**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
MOD	**Remainder after integer division**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
SHL	**Logical shift left**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
SHR	**Logical shift right**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
NOT	**Bitwise negation.** The expression must be an absolute immediate value, and the result is an absolute immediate value.
AND	**Bitwise AND**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
OR	**Bitwise OR**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
XOR	**Bitwise exclusive OR**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.

See Also

Using Inline Assembly Code (see page 610)

Understanding Assembler Syntax (see page 610)

Assembly Procedures and Functions (see page 623)

3.1.3.5.4 Assembly Procedures and Functions (Win32 Only)

You can write complete procedures and functions using inline assembly language code, without including a `begin...end` statement. This topic covers these issues:

- Compiler Optimizations.
- Function Results.

The inline assembler is available only on the Win32 Delphi compiler.

Compiler Optimizations

An example of the type of function you can write is as follows:

```
function LongMul(X, Y: Integer): Longint;
  asm
    MOV     EAX,X
                IMUL Y
        end;
```

The compiler performs several optimizations on these routines:

- No code is generated to copy value parameters into local variables. This affects all string-type value parameters and other value parameters whose size isn't 1, 2, or 4 bytes. Within the routine, such parameters must be treated as if they were **var** parameters.
- Unless a function returns a string, variant, or interface reference, the compiler doesn't allocate a function result variable; a

reference to the **@Result** symbol is an error. For strings, variants, and interfaces, the caller always allocates an **@Result** pointer.

- The compiler only generates stack frames for nested routines, for routines that have local parameters, or for routines that have parameters on the stack.
- **Locals** is the size of the local variables and **Params** is the size of the parameters. If both **Locals** and **Params** are zero, there is no entry code, and the exit code consists simply of a RET instruction.

The automatically generated entry and exit code for the routine looks like this:

```
PUSH            EBP                                 ;Present if Locals <> 0 or Params <> 0
MOV             EBP,ESP                 ;Present if Locals <> 0 or Params <> 0
SUB             ESP,Locals      ;Present if Locals <> 0
.
.
.
MOV             ESP,EBP                 ;Present if Locals <> 0
POP             EBP                                 ;Present if Locals <> 0 or Params <> 0
RET             Params                      ;Always present
```

If locals include variants, long strings, or interfaces, they are initialized to zero but not finalized.

Function Results

Assembly language functions return their results as follows.

- Ordinal values are returned in AL (8-bit values), AX (16-bit values), or EAX (32-bit values).
- Real values are returned in ST(0) on the coprocessor's register stack. (**Currency** values are scaled by 10000.)
- Pointers, including long strings, are returned in EAX.
- Short strings and variants are returned in the temporary location pointed to by `@Result`.

See Also

Using Inline Assembly Code (see page 610)

Understanding Assembler Syntax (see page 610)

Assembly Expressions (see page 616)

3.1.3.6 Object Interfaces

This section describes the use of interfaces in Delphi.

Topics

Name	Description
Object Interfaces (see page 625)	An object interface, or simply interface, defines methods that can be implemented by a class. Interfaces are declared like classes, but cannot be directly instantiated and do not have their own method definitions. Rather, it is the responsibility of any class that supports an interface to provide implementations for the interface's methods. A variable of an interface type can reference an object whose class implements that interface; however, only methods declared in the interface can be called using such a variable. Interfaces offer some of the advantages of multiple inheritance without the semantic difficulties. They are also essential for using... more (see page 625)
Implementing Interfaces (see page 627)	Once an interface has been declared, it must be implemented in a class before it can be used. The interfaces implemented by a class are specified in the class's declaration, after the name of the class's ancestor.
Interface References (see page 631)	If you declare a variable of an interface type, the variable can reference instances of any class that implements the interface. These topics describe Interface references and related topics.

Automation Objects (Win32 Only) (see page 633)	An object whose class implements the IDispatch interface (declared in the `System` unit) is an Automation object. Use variants to access Automation objects. When a variant references an Automation object, you can call the object's methods and read or write to its properties through the variant. To do this, you must include `ComObj` in the **uses** clause of one of your units or your program or library.

3.1.3.6.1 Object Interfaces

An object interface, or simply interface, defines methods that can be implemented by a class. Interfaces are declared like classes, but cannot be directly instantiated and do not have their own method definitions. Rather, it is the responsibility of any class that supports an interface to provide implementations for the interface's methods. A variable of an interface type can reference an object whose class implements that interface; however, only methods declared in the interface can be called using such a variable.

Interfaces offer some of the advantages of multiple inheritance without the semantic difficulties. They are also essential for using distributed object models (such as SOAP). Using a distributed object model, custom objects that support interfaces can interact with objects written in C++, Java, and other languages.

Interface Types

Interfaces, like classes, can be declared only in the outermost scope of a program or unit, not in a procedure or function declaration. An interface type declaration has the form

```
type interfaceName = interface (ancestorInterface)
['{GUID}']
memberList
end;
```

where *(ancestorInterface)* and *['{GUID}']* are optional for .NET interfaces.

Warning: Though the ancestor interface and GUID specification are optional for .NET interfaces, they are required to support Win32 COM interoperability. If your interface is to be accessed through COM, be sure to specify the ancestor interface and GUID.

In most respects, interface declarations resemble class declarations, but the following restrictions apply.

- The *memberList* can include only methods and properties. Fields are not allowed in interfaces.
- Since an interface has no fields, property **read** and **write** specifiers must be methods.
- All members of an interface are public. Visibility specifiers and storage specifiers are not allowed. (But an array property can be declared as **default**.)
- Interfaces have no constructors or destructors. They cannot be instantiated, except through classes that implement their methods.
- Methods cannot be declared as **virtual**, **dynamic**, **abstract**, or **override**. Since interfaces do not implement their own methods, these designations have no meaning.

Here is an example of an interface declaration:

```
type
IMalloc = interface(IInterface)
['{00000002-0000-0000-C000-000000000046}']
function Alloc(Size: Integer): Pointer; stdcall;
function Realloc(P: Pointer; Size: Integer): Pointer; stdcall;
procedure Free(P: Pointer); stdcall;
function GetSize(P: Pointer): Integer; stdcall;
function DidAlloc(P: Pointer): Integer; stdcall;
procedure HeapMinimize; stdcall;
end;
```

In some interface declarations, the **interface** reserved word is replaced by **dispinterface**. This construction (along with the **dispid**, **readonly**, and **writeonly** directives) is platform-specific and is not used in Linux programming.

IInterface and Inheritance

An interface, like a class, inherits all of its ancestors' methods. But interfaces, unlike classes, do not implement methods. What an interface inherits is the obligation to implement methods, an obligation that is passed onto any class supporting the interface.

The declaration of an interface can specify an ancestor interface. If no ancestor is specified, the interface is a direct descendant of `IInterface`, which is defined in the `System` unit and is the ultimate ancestor of all other interfaces. On Win32, `IInterface` declares three methods: `QueryInterface`, `_AddRef`, and `_Release`. These methods are not present on the .NET platform, and you do not need to implement them.

Note: `IInterface` is equivalent to `IUnknown`. You should generally use `IInterface` for platform independent applications and reserve the use of `IUnknown` for specific programs that include Win32 dependencies.

`QueryInterface` provides the means to obtain a reference to the different interfaces that an object supports. `_AddRef` and `_Release` provide lifetime memory management for interface references. The easiest way to implement these methods is to derive the implementing class from the `System` unit's TInterfacedObject. It is also possible to dispense with any of these methods by implementing it as an empty function; COM objects, however, must be managed through `_AddRef` and `_Release`.

Warning: Though `QueryInterface`, `_AddRef`, and `_Release` are optional for .NET interfaces, they are required to support Win32 COM interoperability. If your interface is to be accessed through COM, be sure to implement these methods.

Interface Identification

An interface declaration can specify a globally unique identifier (GUID), represented by a string literal enclosed in brackets immediately preceding the member list. The GUID part of the declaration must have the form

['{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}']

where each x is a hexadecimal digit (0 through 9 or A through F). The Type Library editor automatically generates GUIDs for new interfaces. You can also generate GUIDs by pressing `Ctrl+Shift+G` in the code editor.

A GUID is a 16-byte binary value that uniquely identifies an interface. If an interface has a GUID, you can use interface querying to get references to its implementations.

Note: GUIDs are not required for interfaces in the .NET framework. They are only used for COM interoperability.

The TGUID and PGUID types, declared in the `System` unit, are used to manipulate GUIDs.

```
type
PGUID = ^TGUID;
TGUID = packed record
D1: Longword;
D2: Word;
D3: Word;
D4: array[0..7] of Byte;
end;
```

On the .NET platform, you can tag an interface as described above (i.e. following the interface declaration). However, if you use the traditional Delphi syntax, the first square bracket construct following the interface declaration is taken as a GUID specifier - not as a .NET attribute. (Note that .NET attributes always apply to the *next* symbol, not the previous one.) You can also associate a GUID with an interface using the .NET Guid custom attribute. In this case you would use the .NET style syntax, placing the attribute immediately before the interface declaration.

When you declare a typed constant of type TGUID, you can use a string literal to specify its value. For example,

```
const IID_IMalloc: TGUID = '{00000002-0000-0000-C000-000000000046}';
```

In procedure and function calls, either a GUID or an interface identifier can serve as a value or constant parameter of type TGUID. For example, given the declaration

```
function Supports(Unknown: IInterface; const IID: TGUID): Boolean;
```

Supports can be called in either of two ways

```
if Supports(Allocator, IMalloc) then ...
```

or

```
if Supports(Allocator, IID_IMalloc) then ...
```

Calling Conventions for Interfaces

The default calling convention for interface methods is **register**, but interfaces shared among modules (especially if they are written in different languages) should declare all methods with **stdcall**. On Win32, you can use **safecall** to implement methods of dual interfaces.

Interface Properties

Properties declared in an interface are accessible only through expressions of the interface type; they cannot be accessed through class-type variables. Moreover, interface properties are visible only within programs where the interface is compiled.

In an interface, property **read** and **write** specifiers must be methods, since fields are not available.

Forward Declarations

An interface declaration that ends with the reserved word **interface** and a semicolon, without specifying an ancestor, GUID, or member list, is a forward declaration. A forward declaration must be resolved by a defining declaration of the same interface within the same type declaration section. In other words, between a forward declaration and its defining declaration, nothing can occur except other type declarations.

Forward declarations allow mutually dependent interfaces. For example,

```
type
     IControl = interface;
     IWindow = interface
     ['{00000115-0000-0000-C000-000000000044}']
     function GetControl(Index: Integer): IControl;
.
.
.
     end;
     IControl = interface
     ['{00000115-0000-0000-C000-000000000049}']
     function GetWindow: IWindow;
     .
  .
  .
     end;
```

Mutually derived interfaces are not allowed. For example, it is not legal to derive `IWindow` from `IControl` and also derive `IControl` from `IWindow`.

See Also

Implementing Interfaces (↗ see page 627)

Interface References (↗ see page 631)

Automation Objects (↗ see page 633)

3.1.3.6.2 Implementing Interfaces

Once an interface has been declared, it must be implemented in a class before it can be used. The interfaces implemented by a class are specified in the class's declaration, after the name of the class's ancestor.

Class Declarations

Such declarations have the form

```
type className = class (ancestorClass, interface1, ..., interfacen)
    memberList
end;
```

For example,

```
type
    TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    .
    .
    .
end;
```

declares a class called `TMemoryManager` that implements the `IMalloc` and `IErrorInfo` interfaces. When a class implements an interface, it must implement (or inherit an implementation of) each method declared in the interface.

Here is the (Win32) declaration of TInterfacedObject in the `System` unit. On the .NET platform, TInterfacedObject is an alias for TObject.

```
      type
        TInterfacedObject = class(TObject, IInterface)
        protected
        FRefCount: Integer;
        function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
        function _AddRef: Integer; stdcall;
        function _Release: Integer; stdcall;
        public
        procedure AfterConstruction; override;
        procedure BeforeDestruction; override;
        class function NewInstance: TObject; override;
        property RefCount: Integer read FRefCount;
     end;
```

TInterfacedObject implements the IInterface interface. Hence TInterfacedObject declares and implements each of the three IInterface methods.

Classes that implement interfaces can also be used as base classes. (The first example above declares `TMemoryManager` as a direct descendent of TInterfacedObject.) On the Win32 platform, every interface inherits from IInterface, and a class that implements interfaces must implement the `QueryInterface`, `_AddRef`, and `_Release` methods. The `System` unit's TInterfacedObject implements these methods and is thus a convenient base from which to derive other classes that implement interfaces. On the .NET platform, IInterface does not declare these methods, and you do not need to implement them.

When an interface is implemented, each of its methods is mapped onto a method in the implementing class that has the same result type, the same calling convention, the same number of parameters, and identically typed parameters in each position. By default, each interface method is mapped to a method of the same name in the implementing class.

Method Resolution Clause

You can override the default name-based mappings by including method resolution clauses in a class declaration. When a class implements two or more interfaces that have identically named methods, use method resolution clauses to resolve the naming conflicts.

A method resolution clause has the form

```
procedure interface.interfaceMethod = implementingMethod;
```

or

```
function interface.interfaceMethod = implementingMethod;
```

where *implementingMethod* is a method declared in the class or one of its ancestors. The *implementingMethod* can be a method declared later in the class declaration, but cannot be a private method of an ancestor class declared in another module.

For example, the class declaration

```
type
    TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    function IMalloc.Alloc = Allocate;
    procedure IMalloc.Free = Deallocate;
    .
    .
    .
end;
```

maps `IMalloc`'s `Alloc` and `Free` methods onto `TMemoryManager`'s `Allocate` and `Deallocate` methods.

A method resolution clause cannot alter a mapping introduced by an ancestor class.

Changing Inherited Implementations

Descendant classes can change the way a specific interface method is implemented by overriding the implementing method. This requires that the implementing method be virtual or dynamic.

A class can also reimplement an entire interface that it inherits from an ancestor class. This involves relisting the interface in the descendant class' declaration. For example,

```
type
    IWindow = interface
    ['{00000115-0000-0000-C000-000000000146}']
    procedure Draw;
    .
    .
    .
end;
    TWindow = class(TInterfacedObject, IWindow)// TWindow implements IWindow
    procedure Draw;
    .
    .
    .
end;
TFrameWindow = class(TWindow, IWindow)// TFrameWindow reimplements IWindow
    procedure Draw;
    .
    .
    .
end;
```

Reimplementing an interface hides the inherited implementation of the same interface. Hence method resolution clauses in an ancestor class have no effect on the reimplemented interface.

Implementing Interfaces by Delegation (Win32 only)

On the Win32 platform, the **implements** directive allows you to delegate implementation of an interface to a property in the implementing class. For example,

```
property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
```

declares a property called `MyInterface` that implements the interface `IMyInterface`.

The **implements** directive must be the last specifier in the property declaration and can list more than one interface, separated by commas. The delegate property

- must be of a class or interface type.
- cannot be an array property or have an index specifier.
- must have a **read** specifier. If the property uses a **read** method, that method must use the default **register** calling convention and cannot be dynamic (though it can be virtual) or specify the **message** directive.

The class you use to implement the delegated interface should derive from `TAggregationObject`.

Note: Due to restrictions imposed by the CLR, the implements directive is not supported on the .NET platform.

Delegating to an Interface-Type Property (Win32 only)

If the delegate property is of an interface type, that interface, or an interface from which it derives, must occur in the ancestor list of the class where the property is declared. The delegate property must return an object whose class completely implements the interface specified by the **implements** directive, and which does so without method resolution clauses. For example,

```
type
 IMyInterface = interface
 procedure P1;
    procedure P2;
end;
TMyClass = class(TObject, IMyInterface)
 FMyInterface: IMyInterface;
 property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
end;
var
 MyClass: TMyClass;
 MyInterface: IMyInterface;
begin
 MyClass := TMyClass.Create;
 MyClass.FMyInterface := ...// some object whose class implements IMyInterface
 MyInterface := MyClass;
 MyInterface.P1;
end;
```

Delegating to a Class-Type Property (Win32 only)

If the delegate property is of a class type, that class and its ancestors are searched for methods implementing the specified interface before the enclosing class and its ancestors are searched. Thus it is possible to implement some methods in the class specified by the property, and others in the class where the property is declared. Method resolution clauses can be used in the usual way to resolve ambiguities or specify a particular method. An interface cannot be implemented by more than one class-type property. For example,

```
type
 IMyInterface = interface
 procedure P1;
 procedure P2;
end;
TMyImplClass = class
 procedure P1;
 procedure P2;
end;
 TMyClass = class(TInterfacedObject, IMyInterface)
 FMyImplClass: TMyImplClass;
 property MyImplClass: TMyImplClass read FMyImplClass implements IMyInterface;
 procedure IMyInterface.P1 = MyP1;
 procedure MyP1;
end;
 procedure TMyImplClass.P1;
 .
    .
    .
 procedure TMyImplClass.P2;
 .
    .
 .
 procedure TMyClass.MyP1;
    .
    .
    .
var
 MyClass: TMyClass;
 MyInterface: IMyInterface;
begin
 MyClass := TMyClass.Create;
 MyClass.FMyImplClass := TMyImplClass.Create;
 MyInterface := MyClass;
 MyInterface.P1;    // calls TMyClass.MyP1;
 MyInterface.P2;    // calls TImplClass.P2;
end;
```

See Also

Object Interfaces (see page 625)

Interface References (see page 631)

Automation Objects (see page 633)

3.1.3.6.3 Interface References

If you declare a variable of an interface type, the variable can reference instances of any class that implements the interface. These topics describe Interface references and related topics.

Implementing Interface References

Interface reference variables allow you to call interface methods without knowing at compile time where the interface is implemented. But they are subject to the following:

- An interface-type expression gives you access only to methods and properties declared in the interface, not to other members of the implementing class.
- An interface-type expression cannot reference an object whose class implements a descendant interface, unless the class (or one that it inherits from) explicitly implements the ancestor interface as well.

For example,

```
type
    IAncestor = interface
end;
IDescendant = interface(IAncestor)
    procedure P1;
end;
TSomething = class(TInterfacedObject, IDescendant)
    procedure P1;
    procedure P2;
end;
    .
    .
    .
var
    D: IDescendant;
    A: IAncestor;
begin
    D := TSomething.Create;  // works!
    A := TSomething.Create;  // error
    D.P1;  // works!
    D.P2;  // error
end;
```

In this example, A is declared as a variable of type `IAncestor`. Because `TSomething` does not list IAncestor among the interfaces it implements, a `TSomething` instance cannot be assigned to A. But if we changed `TSomething`'s declaration to

```
TSomething = class(TInterfacedObject, IAncestor, IDescendant)
.
```

```
.
.
```

the first error would become a valid assignment. D is declared as a variable of type `IDescendant`. While D references an instance of `TSomething`, we cannot use it to access `TSomething's` P2 method, since P2 is not a method of `IDescendant`. But if we changed D's declaration to

```
D: TSomething;
```

the second error would become a valid method call.

On the Win32 platform, interface references are typically managed through reference-counting, which depends on the `_AddRef` and `_Release` methods inherited from IInterface. These methods, and reference counting in general, are not applicable on the .NET platform, which is a garbage collected environment. Using the default implementation of reference counting, when an object is referenced only through interfaces, there is no need to destroy it manually; the object is automatically destroyed when the last reference to it goes out of scope. Some classes implement interfaces to bypass this default lifetime management, and some hybrid objects use reference counting only when the object does not have an owner.

Global interface-type variables can be initialized only to **nil**.

To determine whether an interface-type expression references an object, pass it to the standard function **Assigned**.

Interface Assignment Compatibility

Variables of a given class type are assignment-compatible with any interface type implemented by the class. Variables of an interface type are assignment-compatible with any ancestor interface type. The value **nil** can be assigned to any interface-type variable.

An interface-type expression can be assigned to a variant. If the interface is of type `IDispatch` or a descendant, the variant receives the type code `varDispatch`. Otherwise, the variant receives the type code `varUnknown`.

A variant whose type code is `varEmpty`, `varUnknown`, or `varDispatch` can be assigned to an `IInterface` variable. A variant whose type code is varEmpty or varDispatch can be assigned to an `IDispatch` variable.

Interface Typecasts

An interface-type expression can be cast to Variant. If the interface is of type `IDispatch` or a descendant, the resulting variant has the type code `varDispatch`. Otherwise, the resulting variant has the type code `varUnknown`.

A variant whose type code is `varEmpty`, `varUnknown`, or `varDispatch` can be cast to `IInterface`. A variant whose type code is `varEmpty` or `varDispatch` can be cast to `IDispatch`.

Interface Querying

You can use the **as** operator to perform checked interface typecasts. This is known as interface querying, and it yields an interface-type expression from an object reference or from another interface reference, based on the actual (runtime) type of the object. An interface query has the form

```
object as interface
```

where object is an expression of an interface or variant type or denotes an instance of a class that implements an interface, and interface is any interface declared with a GUID.

An interface query returns **nil** if object is **nil**. Otherwise, it passes the GUID of interface to the `QueryInterface` method in object, raising an exception unless `QueryInterface` returns zero. If `QueryInterface` returns zero (indicating that object's class implements interface), the interface query returns an interface reference to object.

See Also

Object Interfaces (see page 625)

Implementing Interfaces (see page 627)

Automation Objects (see page 633)

3.1.3.6.4 Automation Objects (Win32 Only)

An object whose class implements the IDispatch interface (declared in the `System` unit) is an Automation object.

Use variants to access Automation objects. When a variant references an Automation object, you can call the object's methods and read or write to its properties through the variant. To do this, you must include `ComObj` in the **uses** clause of one of your units or your program or library.

Dispatch Interface Types

Dispatch interface types define the methods and properties that an Automation object implements through IDispatch. Calls to methods of a dispatch interface are routed through IDispatch's Invoke method at runtime; a class cannot implement a dispatch interface.

A dispatch interface type declaration has the form

```
type interfaceName = dispinterface
    ['{GUID}']
    memberList
end;
```

where ['{GUID}'] is optional and memberList consists of property and method declarations. Dispatch interface declarations are similar to regular interface declarations, but they cannot specify an ancestor. For example,

```
type
        IStringsDisp = dispinterface
        ['{EE05DFE2-5549-11D0-9EA9-0020AF3D82DA}']
         property ControlDefault[Index: Integer]: OleVariant dispid 0; default;
         function Count: Integer; dispid 1;
      property Item[Index: Integer]: OleVariant dispid 2;
      procedure Remove(Index: Integer); dispid 3;
      procedure Clear; dispid 4;
      function Add(Item: OleVariant): Integer; dispid 5;
         function _NewEnum: IUnknown; dispid -4;
     end;
```

Dispatch interface methods

Methods of a dispatch interface are prototypes for calls to the Invoke method of the underlying IDispatch implementation. To specify an Automation dispatch ID for a method, include the **dispid** directive in its declaration, followed by an integer constant; specifying an already used ID causes an error.

A method declared in a dispatch interface cannot contain directives other than **dispid**. Parameter and result types must be automatable. In other words, they must be **Byte**, **Currency**, **Real**, **Double**, **Longint**, **Integer**, **Single**, **Smallint**, **AnsiString**, **WideString**, TDateTime, Variant, **OleVariant**, **WordBool**, or any interface type.

Dispatch interface properties

Properties of a dispatch interface do not include access specifiers. They can be declared as **read only** or **write only**. To specify a dispatch ID for a property, include the **dispid** directive in its declaration, followed by an integer constant; specifying an already used ID causes an error. Array properties can be declared as **default**. No other directives are allowed in dispatch-interface property declarations.

Accessing Automation Objects

Automation object method calls are bound at runtime and require no previous method declarations. The validity of these calls is not checked at compile time.

The following example illustrates Automation method calls. The `CreateOleObject` function (defined in `ComObj`) returns an `IDispatch` reference to an Automation object and is assignment-compatible with the variant **Word**.

```
var
```

```
Word: Variant;
 begin
  Word := CreateOleObject('Word.Basic');
  Word.FileNew('Normal');
  Word.Insert('This is the first line'#13);
  Word.Insert('This is the second line'#13);
  Word.FileSaveAs('c:\temp\test.txt', 3);
 end;
```

You can pass interface-type parameters to Automation methods.

Variant arrays with an element type of `varByte` are the preferred method of passing binary data between Automation controllers and servers. Such arrays are subject to no translation of their data, and can be efficiently accessed using the `VarArrayLock` and `VarArrayUnlock` routines.

Automation Object Method-Call Syntax

The syntax of an Automation object method call or property access is similar to that of a normal method call or property access. Automation method calls, however, can use both positional and named parameters. (But some Automation servers do not support named parameters.)

A positional parameter is simply an expression. A named parameter consists of a parameter identifier, followed by the **:=** symbol, followed by an expression. Positional parameters must precede any named parameters in a method call. Named parameters can be specified in any order.

Some Automation servers allow you to omit parameters from a method call, accepting their default values. For example,

```
Word.FileSaveAs('test.doc');
Word.FileSaveAs('test.doc', 6);
Word.FileSaveAs('test.doc',,,'secret');
Word.FileSaveAs('test.doc', Password := 'secret');
Word.FileSaveAs(Password := 'secret', Name := 'test.doc');
```

Automation method call parameters can be of integer, real, string, Boolean, and variant types. A parameter is passed by reference if the parameter expression consists only of a variable reference, and if the variable reference is of type **Byte**, **Smallint**, **Integer**, **Single**, **Double**, **Currency**, TDateTime, **AnsiString**, **WordBool**, or Variant. If the expression is not of one of these types, or if it is not just a variable, the parameter is passed by value. Passing a parameter by reference to a method that expects a value parameter causes COM to fetch the value from the reference parameter. Passing a parameter by value to a method that expects a reference parameter causes an error.

Dual Interfaces

A dual interface is an interface that supports both compile-time binding and runtime binding through Automation. Dual interfaces must descend from IDispatch.

All methods of a dual interface (except from those inherited from `IInterface` and `IDispatch`) must use the **safecall** convention, and all method parameter and result types must be automatable. (The automatable types are **Byte**, **Currency**, **Real**, **Double**, **Real48**, **Integer**, **Single**, **Smallint**, **AnsiString**, **ShortString**, TDateTime, Variant, **OleVariant**, and **WordBool**.)

See Also

Object Interfaces (see page 625)

Implementing Interfaces (see page 627)

Interface References (see page 631)

3.1.3.7 Libraries and Packages

This section describes how to create static and dynamically loadable libraries in Delphi.

Topics

Name	Description
Libraries and Packages (↗ see page 635)	A dynamically loadable library is a dynamic-link library (DLL) on Win32, and an assembly (also a DLL) on the .NET platform. It is a collection of routines that can be called by applications and by other DLLs or shared objects. Like units, dynamically loadable libraries contain sharable code or resources. But this type of library is a separately compiled executable that is linked at runtime to the programs that use it. Delphi programs can call DLLs and assemblies written in other languages, and applications written in other languages can call DLLs or assemblies written in Delphi.
Writing Dynamically Loaded Libraries (↗ see page 637)	The following topics describe elements of writing dynamically loadable libraries, including • The exports clause. • Library initialization code. • Global variables. • Libraries and system variables.
Packages (↗ see page 640)	The following topics describe packages and various issues involved in creating and compiling them. • Package declarations and source files • Naming packages • The requires clause • Avoiding circular package references • Duplicate package references • The contains clause • Avoiding redundant source code uses • Compiling packages • Generated files • Package-specific compiler directives • Package-specific command-line compiler switches

3.1.3.7.1 Libraries and Packages

A dynamically loadable library is a dynamic-link library (DLL) on Win32, and an assembly (also a DLL) on the .NET platform. It is a collection of routines that can be called by applications and by other DLLs or shared objects. Like units, dynamically loadable libraries contain sharable code or resources. But this type of library is a separately compiled executable that is linked at runtime to the programs that use it.

Delphi programs can call DLLs and assemblies written in other languages, and applications written in other languages can call DLLs or assemblies written in Delphi.

Calling Dynamically Loadable Libraries

You can call operating system routines directly, but they are not linked to your application until runtime. This means that the library need not be present when you compile your program. It also means that there is no compile-time validation of attempts to import a routine.

Before you can call routines defined in DLL or assembly, you must import them. This can be done in two ways: by declaring an **external** procedure or function, or by direct calls to the operating system. Whichever method you use, the routines are not linked to your application until runtime.

The Delphi language does not support importing of variables from DLLs or assemblies.

Static Loading

The simplest way to import a procedure or function is to declare it using the **external** directive. For example,

```
procedure DoSomething; external 'MYLIB.DLL';
```

If you include this declaration in a program, `MYLIB.DLL` is loaded once, when the program starts. Throughout execution of the program, the identifier `DoSomething` always refers to the same entry point in the same shared library.

Declarations of imported routines can be placed directly in the program or unit where they are called. To simplify maintenance, however, you can collect **external** declarations into a separate "import unit" that also contains any constants and types required for interfacing with the library. Other modules that use the import unit can call any routines declared in it.

Dynamic Loading

You can access routines in a library through direct calls to Win32 APIs, including `LoadLibrary`, `FreeLibrary`, and `GetProcAddress`. These functions are declared in `Windows.pas`. on Linux, they are implemented for compatibility in SysUtils.pas; the actual Linux OS routines are *dlopen*, *dlclose*, and *dlsym* (all declared in libc; see the man pages for more information). In this case, use procedural-type variables to reference the imported routines.

For example,

```
uses Windows, ...;

type
  TTimeRec = record
    Second: Integer;
    Minute: Integer;
    Hour: Integer;
  end;

  TGetTime = procedure(var Time: TTimeRec);
  THandle = Integer;

  var
    Time: TTimeRec;
    Handle: THandle;
    GetTime: TGetTime;
                .
                .
                .
  begin
    Handle := LoadLibrary('libraryname');
    if Handle <> 0 then
    begin
     @GetTime := GetProcAddress(Handle, 'GetTime');
     if @GetTime <> nil then
      begin
      GetTime(Time);
             with Time do
                WriteLn('The time is ', Hour, ':', Minute, ':', Second);
     end;
     FreeLibrary(Handle);
    end;
  end;
```

When you import routines this way, the library is not loaded until the code containing the call to `LoadLibrary` executes. The library is later unloaded by the call to `FreeLibrary`. This allows you to conserve memory and to run your program even when some of the libraries it uses are not present.

See Also

Writing Dynamically Loaded Libraries (see page 637)

Packages (↗ see page 640)

3.1.3.7.2 Writing Dynamically Loaded Libraries

The following topics describe elements of writing dynamically loadable libraries, including

- The exports clause.
- Library initialization code.
- Global variables.
- Libraries and system variables.

Using Export Clause in Libraries

The main source for a dynamically loadable library is identical to that of a program, except that it begins with the reserved word **library** (instead of **program**).

Only routines that a library explicitly exports are available for importing by other libraries or programs. The following example shows a library with two exported functions, `Min` and `Max`.

```
library MinMax;
    function Min(X, Y: Integer): Integer; stdcall;
 begin
   if X < Y then Min := X else Min := Y;
 end;
 function Max(X, Y: Integer): Integer; stdcall;
 begin
   if X > Y then Max := X else Max := Y;
 end;
   exports
     Min,
   Max;
 begin
 end.
```

If you want your library to be available to applications written in other languages, it's safest to specify **stdcall** in the declarations of exported functions. Other languages may not support Delphi's default **register** calling convention.

Libraries can be built from multiple units. In this case, the library source file is frequently reduced to a **uses** clause, an **exports** clause, and the initialization code. For example,

```
library Editors;
  uses EdInit, EdInOut, EdFormat, EdPrint;
  exports
    InitEditors,
    DoneEditors name Done,
    InsertText name Insert,
    DeleteSelection name Delete,
    FormatSelection,
    PrintSelection name Print,
    .
                .
                .
    SetErrorHandler;
  begin
    InitLibrary;
  end.
```

You can put **exports** clauses in the **interface** or **implementation** section of a unit. Any library that includes such a unit in its **uses** clause automatically exports the routines listed the unit's **exports** clauses without the need for an **exports** clause of its own.

The directive **local**, which marks routines as unavailable for export, is platform-specific and has no effect in Windows programming.

On Linux, the local directive provides a slight performance optimization for routines that are compiled into a library but are not exported. This directive can be specified for stand-alone procedures and functions, but not for methods. A routine declared with **local**for example,

```
function Contraband(I: Integer): Integer; local;
```

does not refresh the EBX register and hence

- cannot be exported from a library.
- cannot be declared in the interface section of a unit.
- cannot have its address taken or be assigned to a procedural-type variable.
- if it is a pure assembler routine, cannot be called from another unit unless the caller sets up EBX.

A routine is exported when it is listed in an **exports** clause, which has the form

```
exports entry1, ..., entryn;
```

where each entry consists of the name of a procedure, function, or variable (which must be declared prior to the **exports** clause), followed by a parameter list (only if exporting a routine that is overloaded), and an optional **name** specifier. You can qualify the procedure or function name with the name of a unit.

(Entries can also include the directive **resident**, which is maintained for backward compatibility and is ignored by the compiler.)

On the Win32 platform, an **index** specifier consists of the directive **index** followed by a numeric constant between 1 and 2,147,483,647. (For more efficient programs, use low index values.) If an entry has no **index** specifier, the routine is automatically assigned a number in the export table.

Note: Use of index

specifiers, which are supported for backward compatibility only, is discouraged and may cause problems for other development tools. A **name** specifier consists of the directive **name** followed by a string constant. If an entry has no **name** specifier, the routine is exported under its original declared name, with the same spelling and case. Use a **name** clause when you want to export a routine under a different name. For example,

```
exports
DoSomethingABC name 'DoSomething';
```

When you export an overloaded function or procedure from a dynamically loadable library, you must specify its parameter list in the **exports** clause. For example,

```
exports
Divide(X, Y: Integer) name 'Divide_Ints',
Divide(X, Y: Real) name 'Divide_Reals';
```

On Win32, do not include **index** specifiers in entries for overloaded routines.

An **exports** clause can appear anywhere and any number of times in the declaration part of a program or library, or in the **interface** or **implementation** section of a unit. Programs seldom contain an **exports** clause.

Library Initialization Code

The statements in a library's block constitute the library's initialization code. These statements are executed once every time the library is loaded. They typically perform tasks like registering window classes and initializing variables. Library initialization code can also install an entry point procedure using the `DllProc` variable. The `DllProc` variable is similar to an exit procedure, which is described in Exit procedures; the entry point procedure executes when the library is loaded or unloaded.

Library initialization code can signal an error by setting the `ExitCode` variable to a nonzero value. ExitCode is declared in the `System` unit and defaults to zero, indicating successful initialization. If a library's initialization code sets `ExitCode` to another value, the library is unloaded and the calling application is notified of the failure. Similarly, if an unhandled exception occurs during execution of the initialization code, the calling application is notified of a failure to load the library.

Here is an example of a library with initialization code and an entry point procedure.

```
library Test;
var
 SaveDllProc: Pointer;
 procedure LibExit(Reason: Integer);
begin
 if Reason = DLL_PROCESS_DETACH then
  begin
    .
                .   // library exit code
                .
  end;
   SaveDllProc(Reason);     // call saved entry point procedure
  end;
 begin
    .
                .             // library initialization code
                .
  SaveDllProc := DllProc;       // save exit procedure chain
  DllProc := @LibExit;      // install LibExit exit procedure
 end.
```

`DllProc` is called when the library is first loaded into memory, when a thread starts or stops, or when the library is unloaded. The initialization parts of all units used by a library are executed before the library's initialization code, and the finalization parts of those units are executed after the library's entry point procedure.

Global Variables in a Library

Global variables declared in a shared library cannot be imported by a Delphi application.

A library can be used by several applications at once, but each application has a copy of the library in its own process space with its own set of global variables. For multiple libraries - or multiple instances of a library - to share memory, they must use memory-mapped files. Refer to the your system documentation for further information.

Libraries and System Variables

Several variables declared in the `System` unit are of special interest to those programming libraries. Use IsLibrary to determine whether code is executing in an application or in a library; IsLibrary is always **False** in an application and **True** in a library. During a library's lifetime, `HInstance` contains its instance handle. `CmdLine` is always **nil** in a library.

The DLLProc variable allows a library to monitor calls that the operating system makes to the library entry point. This feature is normally used only by libraries that support multithreading. DLLProc is available on both Windows and Linux but its use differs on each. On Win32, `DLLProc` is used in multithreading applications.; on Linux, it is used to determine when your library is being unloaded. You should use finalization sections, rather than exit procedures, for all exit behavior.

To monitor operating-system calls, create a callback procedure that takes a single integer parameter, for example,

```
procedure DLLHandler(Reason: Integer);
```

and assign the address of the procedure to the `DLLProc` variable. When the procedure is called, it passes to it one of the following values.

DLL_PROCESS_DETACH	Indicates that the library is detaching from the address space of the calling process as a result of a clean exit or a call to `FreeLibrary`.
DLL_PROCESS_ATTACH	Indicates that the library is attaching to the address space of the calling process as the result of a call to `LoadLibrary`.
DLL_THREAD_ATTACH	Indicates that the current process is creating a new thread.
DLL_THREAD_DETACH	Indicates that a thread is exiting cleanly.

In the body of the procedure, you can specify actions to take depending on which parameter is passed to the procedure.

Exceptions and Runtime Errors in Libraries

When an exception is raised but not handled in a dynamically loadable library, it propagates out of the library to the caller. If the calling application or library is itself written in Delphi, the exception can be handled through a normal **try...except** statement.

On Win32, if the calling application or library is written in another language, the exception can be handled as an operating-system exception with the exception code `$0EEDFADE`. The first entry in the `ExceptionInformation` array of the operating-system exception record contains the exception address, and the second entry contains a reference to the Delphi exception object.

Generally, you should not let exceptions escape from your library. Delphi exceptions map to the OS exception model (including the .NET exception model)..

If a library does not use the `SysUtils` unit, exception support is disabled. In this case, when a runtime error occurs in the library, the calling application terminates. Because the library has no way of knowing whether it was called from a Delphi program, it cannot invoke the application's exit procedures; the application is simply aborted and removed from memory.

Shared-Memory Manager (Win32 Only)

On Win32, if a DLL exports routines that pass long strings or dynamic arrays as parameters or function results (whether directly or nested in records or objects), then the DLL and its client applications (or DLLs) must all use the **ShareMem** unit. The same is true if one application or DLL allocates memory with `New` or `GetMem` which is deallocated by a call to **Dispose** or **FreeMem** in another module. **ShareMem** should always be the first unit listed in any program or library **uses** clause where it occurs.

ShareMem is the interface unit for the `BORLANDMM.DLL` memory manager, which allows modules to share dynamically allocated memory. `BORLANDMM.DLL` must be deployed with applications and DLLs that use **ShareMem**. When an application or DLL uses **ShareMem**, its memory manager is replaced by the memory manager in `BORLANDMM.DLL`.

See Also

Libraries and Packages (see page 635)

Packages (see page 640)

3.1.3.7.3 Packages

The following topics describe packages and various issues involved in creating and compiling them.

- Package declarations and source files
- Naming packages
- The requires clause
- Avoiding circular package references
- Duplicate package references
- The contains clause
- Avoiding redundant source code uses
- Compiling packages
- Generated files
- Package-specific compiler directives
- Package-specific command-line compiler switches

Understanding Packages

A package is a specially compiled library used by applications, the IDE, or both. Packages allow you to rearrange where code resides without affecting the source code. This is sometimes referred to as *application partitioning*.

Runtime packages provide functionality when a user runs an application. Design-time packages are used to install components in the IDE and to create special property editors for custom components. A single package can function at both design time and runtime, and design-time packages frequently work by referencing runtime packages in their **requires** clauses.

On Win32, package files end with the `.bpl` (Borland package library) extension. On the .NET platform, packages are .NET assemblies, and end with an extension of `.dll`

Ordinarily, packages are loaded statically when an applications starts. But you can use the `LoadPackage` and `UnloadPackage` routines (in the `SysUtils` unit) to load packages dynamically.

Note: When an application utilizes packages, the name of each packaged unit still must appear in the uses

clause of any source file that references it.

Package Declarations and Source Files

Each package is declared in a separate source file, which should be saved with the `.dpk` extension to avoid confusion with other files containing Delphi code. A package source file does not contain type, data, procedure, or function declarations. Instead, it contains:

- a name for the package.
- a list of other packages required by the new package. These are packages to which the new package is linked.
- a list of unit files contained by, or bound into, the package when it is compiled. The package is essentially a wrapper for these source-code units, which provide the functionality of the compiled package.

A package declaration has the form

package*packageName;*

requiresClause;

containsClause;

end.

where *packageName* is any valid identifier. The *requiresClause* and *containsClause* are both optional. For example, the following code declares the DATAX package.

```
package DATAX;
  requires
  rtl,
  contains Db, DBLocal, DBXpress, ... ;
end.
```

The **requires** clause lists other, external packages used by the package being declared. It consists of the directive **requires**, followed by a comma-delimited list of package names, followed by a semicolon. If a package does not reference other packages, it does not need a **requires** clause.

The **contains** clause identifies the unit files to be compiled and bound into the package. It consists of the directive **contains**, followed by a comma-delimited list of unit names, followed by a semicolon. Any unit name may be followed by the reserved word **in** and the name of a source file, with or without a directory path, in single quotation marks; directory paths can be absolute or relative. For example,

```
contains MyUnit in 'C:\MyProject\MyUnit.pas';
```

Note: Thread-local variables (declared with threadvar

) in a packaged unit cannot be accessed from clients that use the package.

Naming packages

A compiled package involves several generated files. For example, the source file for the package called `DATAX` is `DATAX.DPK`, from which the compiler generates an executable and a binary image called

DATAX.BPL (Win32) or DATAX.DLL (.NET), and DATAX.DCP (Win32) or DATAX.DCPIL (.NET)

DATAX is used to refer to the package in the **requires** clauses of other packages, or when using the package in an application. Package names must be unique within a project.

The requires clause

The **requires** clause lists other, external packages that are used by the current package. It functions like the **uses** clause in a unit file. An external package listed in the **requires** clause is automatically linked at compile time into any application that uses both the current package and one of the units contained in the external package.

If the unit files contained in a package make references to other packaged units, the other packages should be included in the first package's **requires** clause. If the other packages are omitted from the **requires** clause, the compiler loads the referenced units from their .dcu or .dcuil files.

Avoiding circular package references

Packages cannot contain circular references in their **requires** clauses. This means that

- A package cannot reference itself in its own **requires** clause.
- A chain of references must terminate without rereferencing any package in the chain. If package A requires package B, then package B cannot require package A; if package A requires package B and package B requires package C, then package C cannot require package A.

Duplicate package references

The compiler ignores duplicate references in a package's **requires** clause. For programming clarity and readability, however, duplicate references should be removed.

The contains clause

The **contains** clause identifies the unit files to be bound into the package. Do not include file-name extensions in the **contains** clause.

Avoiding redundant source code uses

A package cannot be listed in the **contains** clause of another package or the **uses** clause of a unit.

All units included directly in a package's **contains** clause, or indirectly in the **uses** clauses of those units, are bound into the package at compile time. The units contained (directly or indirectly) in a package cannot be contained in any other packages referenced in **requires** clause of that package.

A unit cannot be contained (directly or indirectly) in more than one package used by the same application.

Compiling Packages

Packages are ordinarily compiled from the IDE using .dpk files generated by the **Project Manager**. You can also compile .dpk files directly from the command line. When you build a project that contains a package, the package is implicitly recompiled, if necessary.

Generated Files

The following table lists the files produced by the successful compilation of a package.

Compiled package files

File extension	Contents
DCP (Win32) or DCPIL (.NET)	A binary image containing a package header and the concatenation of all .dcu (Win32) or .dcuil (.NET) files in the package. A single .dcp or .dcpil file is created for each package. The base name for the file is the base name of the .dpk source file.
BPL (Win32) or DLL (.NET)	The runtime package. This file is a DLL on Win32 with special Borland-specific features. The base name for the package is the base name of the dpk source file.

Package-Specific Compiler Directives

The following table lists package-specific compiler directives that can be inserted into source code.

Package-specific compiler directives

Directive	Purpose
`{$IMPLICITBUILD OFF}`	Prevents a package from being implicitly recompiled later. Use in .dpk files when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.
`{$G-}` or `{$IMPORTEDDATA OFF}`	Disables creation of imported data references. This directive increases memory-access efficiency, but prevents the unit where it occurs from referencing variables in other packages.
`{$WEAKPACKAGEUNIT ON}`	Packages unit weakly.
`{$DENYPACKAGEUNIT ON}`	Prevents unit from being placed in a package.
`{$DESIGNONLY ON}`	Compiles the package for installation in the IDE. (Put in .dpk file.)
`{$RUNONLY ON}`	Compiles the package as runtime only. (Put in .dpk file.)

Including `{$DENYPACKAGEUNIT ON}` in source code prevents the unit file from being packaged. Including `{$G-}` or `{$IMPORTEDDATA OFF}` may prevent a package from being used in the same application with other packages.

Other compiler directives may be included, if appropriate, in package source code.

Package-Specific Command-Line Compiler Switches

The following package-specific switches are available for the command-line compiler.

Package-specific command-line compiler switches

Switch	Purpose
`-$G-`	Disables creation of imported data references. Using this switch increases memory-access efficiency, but prevents packages compiled with it from referencing variables in other packages.
LE path	Specifies the directory where the compiled package file will be placed.
LN path	Specifies the directory where the package dcp or dcpil file will be placed.
LUpackageName [;packageName2;...]	Specifies additional runtime packages to use in an application. Used when compiling a project.
Z	Prevents a package from being implicitly recompiled later. Use when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.

Using the `-$G-` switch may prevent a package from being used in the same application with other packages.

Other command-line options may be used, if appropriate, when compiling packages.

Note: When using the -LU

switch on the .NET platform, you can refer to the package with or without the `.dll` extension. If you omit the `.dll` extension, the compiler will look for the package on the unit search path, and on the package search path. However, if the package specification contains a drive letter or the path separator character, then the compiler will assume the package name is the full file name (including the `.dll` extension). In the latter case, if you specify a full or relative path, but omit the `.dll` extension, the compiler will not be able to locate the package.

See Also

Libraries and Packages (see page 635)

Writing Dynamically Loaded Libraries (see page 637)

3.1.3.8 Memory Management

This section describes memory management issues related to programming in Delphi on Win32, and on .NET.

Topics

Name	Description
Memory Management on the Win32 Platform (see page 644)	The following material describes how memory management on Win32 is handled, and briefly describes memory issues of variables.
Internal Data Formats (see page 645)	The following topics describe the internal formats of Delphi data types.
Memory Management Issues on the .NET Platform (see page 653)	The .NET Common Language Runtime is a garbage-collected environment. This means the programmer is freed (for the most part) from worrying about memory allocation and deallocation. Broadly speaking, after you allocate memory, the CLR determines when it is safe to free that memory. "Safe to free" means that no more references to that memory exist. This topic covers the following memory management issues: • Creating and destroying objects • Unit initialization and finalization sections • Unit initialization and finalization in assemblies and packages

3.1.3.8.1 Memory Management on the Win32 Platform

The following material describes how memory management on Win32 is handled, and briefly describes memory issues of variables.

The Memory Manager (Win32 Only)

The Memory Manager manages all dynamic memory allocations and deallocations in an application. The New, Dispose, GetMem, ReallocMem, and FreeMem standard procedures use the memory manager, and all objects and long strings are allocated through the memory manager.

The Memory Manager is optimized for applications that allocate large numbers of small- to medium-sized blocks, as is typical for object-oriented applications and applications that process string data. The Memory Manager is optimized for efficient operation (high speed and low memory overhead) in single and multi-threaded applications. Other memory managers, such as the implementations of GlobalAlloc, LocalAlloc, and private heap support in Windows, typically do not perform well in such situations, and would slow down an application if they were used directly.

To ensure the best performance, the Memory Manager interfaces directly with the Win32 virtual memory API (the >VirtualAlloc and VirtualFree functions). The Memory Manager supports a user mode address space up to 4GB.

Memory Manager blocks are always rounded upward to a size that is a multiple of 4 bytes, and always include a 4-byte header in which the size of the block and other status bits are stored. The start address of memory blocks are always aligned on at least 8-byte boundaries, or optionally on 16-byte boundaries, which improves performance when addressing them.

The Memory Manager employs an algorithm that anticipates future block reallocations, reducing the performance impact usually associated with such operations. The reallocation algorithm also helps reduce address space fragmentation.

The memory manager provides a sharing mechanism that does not require the use of an external DLL.

The Memory Manager includes reporting functions to help applications monitor their own memory usage and potential memory leaks.

The Memory Manager provides two procedures, GetMemoryManagerState and GetMemoryMap, that allow applications to retrieve memory-manager status information and a detailed map of memory usage.

Variables

Global variables are allocated on the application data segment and persist for the duration of the program. Local variables (declared within procedures and functions) reside in an application's stack. Each time a procedure or function is called, it allocates a set of local variables; on exit, the local variables are disposed of. Compiler optimization may eliminate variables earlier.

On Win32, an application's stack is defined by two values: the minimum stack size and the maximum stack size. The values are controlled through the `$MINSTACKSIZE` and `$MAXSTACKSIZE` compiler directives, and default to 16,384 (16K) and 1,048,576 (1Mb) respectively. An application is guaranteed to have the minimum stack size available, and an application's stack is never allowed to grow larger than the maximum stack size. If there is not enough memory available to satisfy an application's minimum stack requirement, Windows will report an error upon attempting to start the application.

If a Win32 application requires more stack space than specified by the minimum stack size, additional memory is automatically allocated in 4K increments. If allocation of additional stack space fails, either because more memory is not available or because the total size of the stack would exceed the maximum stack size, an **EStackOverflow** exception is raised. (Stack overflow checking is completely automatic. The `$S` compiler directive, which originally controlled overflow checking, is maintained for backward compatibility.)

Dynamic variables created with the GetMem or New procedure are heap-allocated and persist until they are deallocated with FreeMem or Dispose.

Long strings, wide strings, dynamic arrays, variants, and interfaces are heap-allocated, but their memory is managed automatically.

See Also

Internal Data Formats (see page 645)

Memory Management Issues on the .NET Platform (see page 653)

Configuring the Memory Manager (see page 175)

Increasing the Memory Manager Address Space Beyond 2GB (see page 176)

Registering Memory Leaks (see page 178)

Monitoring the Memory Manager (see page 177)

Sharing Memory (see page 178)

3.1.3.8.2 Internal Data Formats

The following topics describe the internal formats of Delphi data types.

Integer Types

The format of an integer-type variable depends on its minimum and maximum bounds.

- If both bounds are within the range 128..127 (Shortint), the variable is stored as a signed byte.
- If both bounds are within the range 0..255 (Byte), the variable is stored as an unsigned byte.
- If both bounds are within the range 32768..32767 (Smallint), the variable is stored as a signed word.
- If both bounds are within the range 0..65535 (Word), the variable is stored as an unsigned word.
- If both bounds are within the range 2147483648..2147483647 (Longint), the variable is stored as a signed double word.
- If both bounds are within the range 0..4294967295 (Longword), the variable is stored as an unsigned double word.
- Otherwise, the variable is stored as a signed quadruple word (Int64).

  Note: a "word" occupies two bytes.

Character Types

On the Win32 platform, **Char**, an **AnsiChar**, or a subrange of a **Char** type is stored as an unsigned byte. A **WideChar** is stored as an unsigned word.

On the .NET platform, a **Char** is equivalent to **WideChar**.

Boolean Types

A **Boolean** type is stored as a **Byte**, a **ByteBool** is stored as a **Byte**, a **WordBool** type is stored as a **Word**, and a **LongBool** is stored as a **Longint**.

A **Boolean** can assume the values 0 (**False**) and 1 (**True**). **ByteBool**, **WordBool**, and **LongBool** types can assume the values 0 (**False**) or nonzero (**True**).

Enumerated Types

An enumerated type is stored as an unsigned byte if the enumeration has no more than 256 values and the type was declared in the `{$Z1}` state (the default). If an enumerated type has more than 256 values, or if the type was declared in the `{$Z2}` state, it is stored as an unsigned word. If an enumerated type is declared in the `{$Z4}` state, it is stored as an unsigned double-word.

Real Types

The real types store the binary representation of a sign (**+** or **-**), an exponent, and a significand. A real value has the form

+/- *significand* * 2exponent

where the *significand* has a single bit to the left of the binary decimal point. (That is, 0 <= *significand* < 2.)

In the figures that follow, the most significant bit is always on the left and the least significant bit on the right. The numbers at the top indicate the width (in bits) of each field, with the left-most items stored at the highest addresses. For example, for a Real48 value, e is stored in the first byte, *f* in the following five bytes, and *s* in the most significant bit of the last byte.

The Real48 type

The following discussion of the **Real48** type applies only to the Win32 platform. The **Real48** type is not supported on the .NET platform.

On the Win32 platform, a 6-byte (48-bit) Real48 number is divided into three fields:

1	39	8
s	f	e

If 0 < *e* <= 255, the value *v* of the number is given by

v = (1)^s * 2^(e129) * (1.f)

If e = 0, then v = 0.

The **Real48** type can't store denormals, NaNs, and infinities. Denormals become zero when stored in a **Real48**, while NaNs and infinities produce an overflow error if an attempt is made to store them in a **Real48**.

The Single type

A 4-byte (32-bit) Single number is divided into three fields

1	8	23
s	e	f

The value *v* of the number is given by

if 0 < e < 255, then v = (1)^s * 2^(e127) * (1.f)

if e = 0 and f <> 0, then v = (1)^s * 2^(126) * (0.f)

if e = 0 and f = 0, then v = (1)^s * 0

if e = 255 and f = 0, then v = (1)^s * Inf

if e = 255 and f <> 0, then v is a NaN

The Double type

An 8-byte (64-bit) Double number is divided into three fields

1	11	52
s	e	f

The value *v* of the number is given by

if 0 < e < 2047, then v = (1)^s * 2^(e1023) * (1.f)

if e = 0 and f <> 0, then v = (1)^s * 2^(1022) * (0.f)

if e = 0 and f = 0, then v = (1)^s * 0

if e = 2047 and f = 0, then v = (1)^s * Inf

if e = 2047 and f <> 0, then v is a NaN

The Extended type

A 10-byte (80-bit) Extended number is divided into four fields:

1	15	1	63
s	e	i	f

The value *v* of the number is given by

if 0 <= e < 32767, then v = (1)^s * 2^(e16383) * (i.f)

if e = 32767 and f = 0, then v = (1)^s * Inf

if e = 32767 and f <> 0, then v is a NaN

Note: On the .NET platform, the Extended type is aliased to Double, and has been deprecated.

The Comp type

An 8-byte (64-bit) Comp number is stored as a signed 64-bit integer.

Note: On the .NET platform, the Comp type is aliased to Int64, and has been deprecated.

The Currency type

An 8-byte (64-bit) Currency number is stored as a scaled and signed 64-bit integer with the four least-significant digits implicitly representing four decimal places.

Pointer Types

A Pointer type is stored in 4 bytes as a 32-bit address. The pointer value **nil** is stored as zero.

Note: On the .NET platform, the size of a pointer will vary at runtime. Therefore, `SizeOf(pointer)` is not a compile-time constant, as it is on the Win32 platform.

Short String Types

A string occupies as many bytes as its maximum length plus one. The first byte contains the current dynamic length of the string, and the following bytes contain the characters of the string.

The length byte and the characters are considered unsigned values. Maximum string length is 255 characters plus a length byte (`string[255]`).

Note: On the .NET platform, the short string type is implemented as an array of unsigned bytes.

Long String Types

A long string variable occupies four bytes of memory which contain a pointer to a dynamically allocated string. When a long string variable is empty (contains a zero-length string), the string pointer is **nil** and no dynamic memory is associated with the string variable. For a nonempty string value, the string pointer points to a dynamically allocated block of memory that contains the string value in addition to a 32-bit length indicator and a 32-bit reference count. The table below shows the layout of a long-string memory block.

Long string dynamic memory layout (Win32 only)

Offset	Contents
-8	32-bit reference-count
-4	length in bytes
0..Length - 1	character string
Length	NULL character

The NULL character at the end of a long string memory block is automatically maintained by the compiler and the built-in string handling routines. This makes it possible to typecast a long string directly to a null-terminated string.

For string constants and literals, the compiler generates a memory block with the same layout as a dynamically allocated string, but with a reference count of -1. When a long string variable is assigned a string constant, the string pointer is assigned the address of the memory block generated for the string constant. The built-in string handling routines know not to attempt to modify blocks that have a reference count of -1.

Note: On the .NET platform, an AnsiString is implemented as an array of unsigned bytes. The information above on string constants and literals does not apply to the .NET platform.

Wide String Types

On Win32, a wide string variable occupies four bytes of memory which contain a pointer to a dynamically allocated string. When a wide string variable is empty (contains a zero-length string), the string pointer is **nil** and no dynamic memory is associated with the string variable. For a nonempty string value, the string pointer points to a dynamically allocated block of memory that contains the string value in addition to a 32-bit length indicator. The table below shows the layout of a wide string memory block on Windows.

Wide string dynamic memory layout (Win32 only)

Offset	Contents
-4	32-bit length indicator (in bytes)
0..Length -1	character string
Length	NULL character

The string length is the number of bytes, so it is twice the number of wide characters contained in the string.

The NULL character at the end of a wide string memory block is automatically maintained by the compiler and the built-in string handling routines. This makes it possible to typecast a wide string directly to a null-terminated string.

On the .NET platform, String and WideString types are implemented using the System.String type. An empty string is not a **nil** pointer, and the string data is not terminated with a null character.

Set Types

A set is a bit array where each bit indicates whether an element is in the set or not. The maximum number of elements in a set is 256, so a set never occupies more than 32 bytes. The number of bytes occupied by a particular set is equal to (*Max* div 8) (*Min* div 8) + 1, where *Max* and *Min* are the upper and lower bounds of the base type of the set. The byte number of a specific element *E* is (*E* div 8) (*Min* div 8) and the bit number within that byte is *E* mod 8, where E denotes the ordinal value of the element. When possible, the compiler stores sets in CPU registers, but a set always resides in memory if it is larger than the generic Integer type or if the program contains code that takes the address of the set.

Note: On the .NET platform, sets containing more than 32 elements are implemented as an array of bytes. The set type in Delphi for .NET is not CLS compliant, therefore other .NET languages cannot use them.

Static Array Types

On the Win32 platform, a static array is stored as a contiguous sequence of variables of the component type of the array. The components with the lowest indexes are stored at the lowest memory addresses. A multidimensional array is stored with the rightmost dimension increasing first.

On the .NET platform, static are implemented using the System.Array type. Memory layout is therefore determined by the System.Array type.

Dynamic Array Types

On the Win32 platform, a dynamic-array variable occupies four bytes of memory which contain a pointer to the dynamically allocated array. When the variable is empty (uninitialized) or holds a zero-length array, the pointer is **nil** and no dynamic memory is associated with the variable. For a nonempty array, the variable points to a dynamically allocated block of memory that contains the array in addition to a 32-bit length indicator and a 32-bit reference count. The table below shows the layout of a dynamic-array memory block.

Dynamic array memory layout (Win32 only)

Offset	Contents
-8	32-bit reference-count
-4	32-bit length indicator (number of elements)
0..Length * (size of element) -1	array elements

On the .NET platform, dynamic arrays and open array parameters are implemented using the System.Array type. As with static arrays, memory layout is therefore determined by the System.Array type.

Record Types

On the .NET platform, field layout in record types is determined at runtime, and can vary depending on the architecture of the target hardware. The following discussion of record alignment applies to the Win32 platform only (**packed** records are supported on the .NET platform, however).

When a record type is declared in the `{$A+}` state (the default), and when the declaration does not include a **packed** modifier, the type is an unpacked record type, and the fields of the record are aligned for efficient access by the CPU. The alignment is controlled by the type of each field and by whether fields are declared together. Every data type has an inherent alignment, which is automatically computed by the compiler. The alignment can be 1, 2, 4, or 8, and represents the byte boundary that a value of the type must be stored on to provide the most efficient access. The table below lists the alignments for all data types.

Type alignment masks (Win32 only)

Type	Alignment
Ordinal types	size of the type (1, 2, 4, or 8)
Real types	2 for *Real48*, 4 for *Single*, 8 for *Double* and *Extended*
Short string types	1
Array types	same as the element type of the array.
Record types	the largest alignment of the fields in the record
Set types	size of the type if 1, 2, or 4, otherwise 1
All other types	determined by the $A directive.

To ensure proper alignment of the fields in an unpacked record type, the compiler inserts an unused byte before fields with an alignment of 2, and up to three unused bytes before fields with an alignment of 4, if required. Finally, the compiler rounds the total size of the record upward to the byte boundary specified by the largest alignment of any of the fields.

If two fields share a common type specification, they are packed even if the declaration does not include the **packed** modifier and the record type is not declared in the `{$A-}` state. Thus, for example, given the following declaration

```
type
  TMyRecord = record
    A, B: Extended;
    C: Extended;
  end;
```

`A` and `B` are packed (aligned on byte boundaries) because they share the same type specification. The compiler pads the structure with unused bytes to ensure that `C` appears on a quadword boundary.

When a record type is declared in the `{$A-}` state, or when the declaration includes the **packed** modifier, the fields of the record are not aligned, but are instead assigned consecutive offsets. The total size of such a packed record is simply the size of all the fields. Because data alignment can change, it's a good idea to pack any record structure that you intend to write to disk or pass in memory to another module compiled using a different version of the compiler.

File Types

The following discussion of file types applies to the Win32 platform only. On the .NET platform, text files are implemented with a class (as opposed to a record). Binary file types (e.g. `File of MyType`) are not supported on the .NET platform.

On the Win32 platform, file types are represented as records. Typed files and untyped files occupy 332 bytes, which are laid out as follows:

```
type
TFileRec = packed record
  Handle: Integer;
```

```
    Mode: word;
    Flags: word;
      case Byte of
          0: (RecSize: Cardinal);
          1: (BufSize: Cardinal;
              BufPos: Cardinal;
              BufEnd: Cardinal;
              BufPtr: PChar;
              OpenFunc: Pointer;
              InOutFunc: Pointer;
              FlushFunc: Pointer;
              CloseFunc: Pointer;
              UserData: array[1..32] of Byte;
              Name: array[0..259] of Char; );
end;
```

Text files occupy 460 bytes, which are laid out as follows:

```
type
        TTextBuf = array[0..127] of Char;
        TTextRec = packed record
                Handle: Integer;
                Mode: word;
                Flags: word;
                BufSize: Cardinal;
                BufPos: Cardinal;
                BufEnd: Cardinal;
                BufPtr: PChar;
                OpenFunc: Pointer;
                InOutFunc: Pointer;
                FlushFunc: Pointer;
                CloseFunc: Pointer;
                UserData: array[1..32] of Byte;
                Name: array[0..259] of Char;
                Buffer: TTextBuf;
end;
```

Handle contains the file's handle (when the file is open).

The `Mode` field can assume one of the values

```
const
    fmClosed = $D7B0;
    fmInput= $D7B1;
    fmOutput = $D7B2;
    fmInOut= $D7B3;
```

where *fmClosed* indicates that the file is closed, *fmInput* and *fmOutput* indicate a text file that has been reset (*fmInput*) or rewritten (*fmOutput*), *fmInOut* indicates a typed or untyped file that has been reset or rewritten. Any other value indicates that the file variable is not assigned (and hence not initialized).

The *UserData* field is available for user-written routines to store data in.

Name contains the file name, which is a sequence of characters terminated by a null character (#0).

For typed files and untyped files, *RecSize* contains the record length in bytes, and the *Private* field is unused but reserved.

For text files, *BufPtr* is a pointer to a buffer of *BufSize* bytes, *BufPos* is the index of the next character in the buffer to read or write, and *BufEnd* is a count of valid characters in the buffer. *OpenFunc*, *InOutFunc*, *FlushFunc*, and *CloseFunc* are pointers to the I/O routines that control the file; see Device functions. Flags determines the line break style as follows:

bit 0 clear	LF line breaks
bit 0 set	CRLF line breaks

All other Flags bits are reserved for future use.

Procedural Types

On the Win32 platform, a procedure pointer is stored as a 32-bit pointer to the entry point of a procedure or function. A method pointer is stored as a 32-bit pointer to the entry point of a method, followed by a 32-bit pointer to an object.

On the .NET platform, procedural types are implemented using the System.MulticastDelegate class types.

Class Types

The following discussion of the internal layout of class types applies to the Win32 platform only. On the .NET platform, class layout is determined at runtime. Runtime type information is obtained using the System.Reflection APIs in the .NET framework.

On the Win32 platform, a class-type value is stored as a 32-bit pointer to an instance of the class, which is called an *object*. The internal data format of an object resembles that of a record. The object's fields are stored in order of declaration as a sequence of contiguous variables. Fields are always aligned, corresponding to an unpacked record type. Any fields inherited from an ancestor class are stored before the new fields defined in the descendant class.

The first 4-byte field of every object is a pointer to the *virtual method table* (VMT) of the class. There is exactly one VMT per class (not one per object); distinct class types, no matter how similar, never share a VMT. VMT's are built automatically by the compiler, and are never directly manipulated by a program. Pointers to VMT's, which are automatically stored by constructor methods in the objects they create, are also never directly manipulated by a program.

The layout of a VMT is shown in the following table. At positive offsets, a VMT consists of a list of 32-bit method pointersone per user-defined virtual method in the class typein order of declaration. Each slot contains the address of the corresponding virtual method's entry point. This layout is compatible with a C++ v-table and with COM. At negative offsets, a VMT contains a number of fields that are internal to Delphi's implementation. Applications should use the methods defined in TObject to query this information, since the layout is likely to change in future implementations of the Delphi language.

Virtual method table layout (Win32 Only)

Offset	Type	Description
-76	Pointer	pointer to virtual method table (or **nil**)
-72	Pointer	pointer to interface table (or **nil**)
-68	Pointer	pointer to Automation information table (or **nil**)
-64	Pointer	pointer to instance initialization table (or **nil**)
-60	Pointer	pointer to type information table (or **nil**)
-56	Pointer	pointer to field definition table (or **nil**)
-52	Pointer	pointer to method definition table (or **nil**)
-48	Pointer	pointer to dynamic method table (or **nil**)
-44	Pointer	pointer to short string containing class name
-40	Cardinal	instance size in bytes
-36	Pointer	pointer to a pointer to ancestor class (or **nil**)
-32	Pointer	pointer to entry point of *SafecallException* method (or **nil**)
-28	Pointer	entry point of *AfterConstruction* method
-24	Pointer	entry point of *BeforeDestruction* method
-20	Pointer	entry point of *Dispatch* method
-16	Pointer	entry point of *DefaultHandler* method
-12	Pointer	entry point of *NewInstance* method
-8	Pointer	entry point of *FreeInstance* method

-4	Pointer	entry point of *Destroy* destructor
0	Pointer	entry point of first user-defined virtual method
4	Pointer	entry point of second user-defined virtual method

Class Reference Types

On the Win32 platform, a class-reference value is stored as a 32-bit pointer to the virtual method table (VMT) of a class.

On the .NET platform, class reference types are implemented using compiler-constructed nested classes inside the class type they support. These implementation details are subject to change in future compiler releases.

Variant Types

The following discussion of the internal layout of variant types applies to the Win32 platform only. On the .NET platform, variants are an alias of System.Object. Variants rely on boxing and unboxing of data into an object wrapper, as well as Delphi helper classes to implement the variant-related RTL functions.

On the Win32 platform, a variant is stored as a 16-byte record that contains a type code and a value (or a reference to a value) of the type given by the code. The `System` and `Variants` units define constants and types for variants.

The **TVarData** type represents the internal structure of a Variant variable (on Windows, this is identical to the Variant type used by COM and the Win32 API). The **TVarData** type can be used in typecasts of Variant variables to access the internal structure of a variable. The *TVarData* record contains the following fields:

- *VType* contains the type code of the variant in the lower twelve bits (the bits defined by the **varTypeMask** constant). In addition, the *varArray* bit may be set to indicate that the variant is an array, and the *varByRef* bit may be set to indicate that the variant contains a reference as opposed to a value.
- The **Reserved1**, **Reserved2**, and **Reserved3** fields are unused.

The contents of the remaining eight bytes of a *TVarData* record depend on the *VType* field as follows:

- If neither the *varArray* nor the *varByRef* bits are set, the variant contains a value of the given type.
- If the *varArray* bit is set, the variant contains a pointer to a *TVarArray* structure that defines an array. The type of each array element is given by the *varTypeMask* bits in the *VType* field.
- If the *varByRef* bit is set, the variant contains a reference to a value of the type given by the *varTypeMask* and varArray bits in the *VType* field.

The *varString* type code is private. Variants containing a *varString* value should never be passed to a non-Delphi function. On Win32, Delphi's Automation support automatically converts *varString* variants to *varOleStr* variants before passing them as parameters to external functions.

See Also

Memory Management on the Win32 Platform (see page 644)

Memory Management Issues on the .NET Platform (see page 653)

Data Types (see page 553)

3.1.3.8.3 Memory Management Issues on the .NET Platform

The .NET Common Language Runtime is a garbage-collected environment. This means the programmer is freed (for the most part) from worrying about memory allocation and deallocation. Broadly speaking, after you allocate memory, the CLR determines when it is safe to free that memory. "Safe to free" means that no more references to that memory exist.

This topic covers the following memory management issues:

- Creating and destroying objects

- Unit initialization and finalization sections
- Unit initialization and finalization in assemblies and packages

Constructors

In Delphi for .NET, a constructor must always call an inherited constructor before it may access or initialize any inherited class members. The compiler generates an error if your constructor code does not call the inherited constructor (a valid situation in Delphi for Win32), but it is important to examine your constructors to make sure that you do not access any inherited class fields, directly or indirectly, before the call to the inherited constructor.

Note: A constructor can initialize fields from its own class, prior to calling the inherited constructor.

Finalization

Every class in the .NET Framework (including VCL.NET classes) inherits a method called Finalize. The garbage collector calls the Finalize method when the memory for the object is about to be freed. Since the method is called by the garbage collector, you have no control over when it is called. The asynchronous nature of finalization is a problem for objects that open resources such as file handles and database connections, because the Finalize method might not be called for some time, leaving these connections open.

To add a finalizer to a class, override the `strict protected` Finalize procedure that is inherited from TObject. The .NET platform places limits on what you can do in a finalizer, because it is called when the garbage collector is cleaning up objects. The finalizer may execute in a different thread than the thread the object was was created in. A finalizer cannot allocate new memory, and cannot make calls outside of itself. If your class has references to other objects, a finalizer can refer to them (that is, their memory is guaranteed not to have been freed yet), but be aware that their state is undefined, as you do not know whether they have been finalized yet.

When a class has a finalizer, the CLR must add newly instantiated objects of the class to the finalization list. Further, objects with finalizers tend to persist in memory longer, as they are not freed when the garbage collector first determines that they are no longer actively referenced. If the object has references to other objects, those objects are also not freed right away (even if they don't have finalizers themselves), but must also persist in memory until the original object is finalized. Therefore, finalizers do impart a fair amount of overhead in terms of memory consumption and execution performance, so they should be used judiciously.

It is a good practice to restrict finalizers to small objects that represent unmanaged resources. Classes that use these resources can then hold a reference to the small object with the finalizer. In this way, big classes, and classes that reference many other classes, do not hoard memory because of a finalizer.

Another good practice is to suppress finalizers when a particular resource has already been released in a destructor. After freeing the resources, you can call SuppressFinalize, which causes the CLR to remove the object from the finalization list. Be careful not to call SuppressFinalize with a **nil** reference, as that causes a runtime exception.

The Dispose Pattern

Another way to free up resources is to implement the *dispose* pattern. Classes adhering to the dispose pattern must implement the .NET interface called IDisposable. IDisposable contains only one method, called Dispose. Unlike the Finalize method, the Dispose method is public. It can be called directly by a user of the class, as opposed to relying on the garbage collector to call it. This gives you back control of freeing resources, but calling Dispose still does not reclaim memory for the object itself - that is still for the garbage collector to do. Note that some classes in the .NET Framework implement both Dispose, and another method such as `Close`. Typically the `Close` method simply calls Dispose, but the extra method is provided because it seems more "natural" for certain classes such as files.

Delphi for .NET classes are free to use the Finalize method for freeing system resources, however the recommended method is to implement the dispose pattern. The Delphi for .NET compiler recognizes a very specific destructor pattern in your class, and implements the IDisposable interface for you. This enables you to continue writing new code for the .NET platform the same way you always have, while allowing much of your existing Win32 Delphi code to run in the garbage collected environment of the CLR.

The compiler recognizes the following specific pattern of a Delphi destructor:

```
TMyClass = class(TObject)
  destructor Destroy; override;
end;
```

Your destructor must fit this pattern exactly:

- The name of the destructor must be `Destroy`.
- The keyword **override** must be specified.
- The destructor cannot take any parameters.

In the compiler's implementation of the dispose pattern, the Free method is written so that if the class implements the IDisposable interface (which it does), then the Dispose method is called, which in turn calls your destructor.

You can still implement the IDisposable interface directly, if you choose. However, the compiler's automatic implementation of the Free-Dispose-Destroy mechanism cannot coexist with your implementation of IDisposable. The two methods of implementing IDisposable are mutually exclusive. You must choose to either implement IDisposable directly, or equip your class with the familiar `destructor Destroy; override` pattern and rely on the compiler to do the rest. The `Free` method will call Dispose in either case, but if you implement Dispose yourself, you must call your destructor yourself. If you want to implement IDisposable yourself, your destructor cannot be called `Destroy`.

Note: You can declare destructors with other names; the compiler only provides the IDisposable implementation when the destructor fits the above pattern.

The Dispose method is not called automatically; the Free method must be called in order for Dispose to be called. If an object is freed by the garbage collector because there are no references to it, but you did not explicitly call Free on the object, the object will be freed, but the destructor will not execute.

Note: When the garbage collector frees the memory used by an object, it also reclaims the memory used by all fields of the object instance as well. This means the most common reason for implementing destructors in Delphi for Win32 - to release allocated memory - no longer applies. However, in most cases, unmanaged resources such as window handles or file handles still need to be released.

To eliminate the possibility of destructors being called more than once, the Delphi for .NET compiler introduces a field called `DisposeCount` into every class declaration. If the class already has a field by this name, the name collision will cause the compiler to produce a syntax error in the destructor.

Unit Initialization and Finalization

On the .NET platform, units that you depend on will be initialized prior to initializing your own unit. However, there is no way to guarantee the order in which units are initialized. Nor is there a way to guarantee when they will be initialized. Be aware of initialization code that depends on another unit's initialization side effects, such as the creation of a file. Such a dependency cannot be made to work reliably on the .NET platform.

Unit finalization is subject to the same constraints and difficulties as the Finalize method of objects. Specifically, unit finalization is asynchronous, and, there no way to determine when it will happen (or if it will happen, though under most circumstances, it will).

Typical tasks performed in a unit finalization include freeing global objects, unregistering objects that are used by other units, and freeing resources. Because .NET is a memory managed environment, the garbage collector will free global objects even if the unit finalization section is not called. The units in an application domain are loaded and unloaded together, so you do not need to worry about unregistering objects. All units that can possibly refer to each other (even in different assemblies) are released at the same time. Since object references do not cross application domains, there is no danger of something keeping a dangling reference to an object type or code that has been unloaded from memory.

Freeing resources (such as file handles or window handles) is the most important consideration in unit finalization. Because unit finalization sections are not guaranteed to be called, you may want to rewrite your code to handle this issue using finalizers rather than relying on the unit finalization.

The main points to keep in mind for unit initialization and finalization on the .NET platform are:

1. The Finalize method is called asynchronously (both for objects, and for units).

2. Finalization and destructors are used to free unmanaged resources such as file handles. You do not need to destroy object member variables; the garbage collector takes care of this for you.
3. Classes should rely on the compiler's implementation of IDisposable, and provide a destructor called `Destroy`.
4. If a class implements IDisposable itself, it cannot have a destructor called `Destroy`.
5. Reference counting is deprecated. Try to use the `destructor Destroy; override;` pattern wherever possible.
6. Unit initialization should not depend on side effects produced by initialization of dependent units.

Unit Initialization Considerations for Assemblies and Dynamically Linked Packages

Under Win32, the Delphi compiler uses the `DllMain` function as a hook from which to execute unit initialization code. No such execution path exists in the .NET environment. Fortunately, other means of implementing unit initialization exist in the .NET Framework. However, the differences in the implementation between Win32 and .NET could impact the order of unit initialization in your application.

The Delphi for .NET compiler uses CLS-compliant class constructors to implement unit initialization hooks. The CLR requires that every object type have a class constructor. These constructors, or type initializers, are guaranteed to be executed *at most* one time. Class constructors are executed at most one time, because in order for the type to be loaded, it must be used. That is, the assembly containing a type will not be loaded until the type is actually used at runtime. If the assembly is never loaded, its unit initialization section will never run.

Circular unit references also impact the unit initialization process. If unit A uses unit B, and unit B then uses unit A in its implementation section, the order of unit initialization is undefined. To fully understand the possibilities, it is helpful to look at the process one step at a time.

1. Unit A's initialization section uses a type from unit B. If this is the first reference to the type, the CLR will load its assembly, triggering the unit initialization of unit B.
2. As a consequence, loading and initializing unit B occurs before unit A's initialization section has completed execution. Note this is a change from how unit initialization works under Win32.
3. Suppose that unit B's initialization is in progress, and that a type from unit A is used. Unit A has not completed initialization, and such a reference could cause an access violation.

The unit initialization should only use types defined within that unit. Using types from outside the unit will impact unit initialization, and could cause an access violation, as noted above.

Unit initialization for DLLs happens automatically; it is triggered when a type within the DLL is referenced. Applications created with other .NET languages can use Delphi for .NET assemblies without concern for the details of unit initialization.

See Also

Memory Management on the Win32 Platform (see page 644)

Internal Data Formats (see page 645)

3.1.3.9 Delphi Overview

This chapter provides a brief introduction to Delphi programs, and program organization.

Topics

Name	Description
Language Overview (↗ see page 657)	Delphi is a high-level, compiled, strongly typed language that supports structured and object-oriented design. Based on Object Pascal, its benefits include easy-to-read code, quick compilation, and the use of multiple unit files for modular programming. Delphi has special features that support CodeGear's component framework and RAD environment. For the most part, descriptions and examples in this language guide assume that you are using CodeGear development tools. Most developers using CodeGear software development tools write and compile their code in the integrated development environment (IDE). CodeGear development tools handle many details of setting up projects and source files, such as maintenance... more (↗ see page 657)

3.1.3.9.1 Language Overview

Delphi is a high-level, compiled, strongly typed language that supports structured and object-oriented design. Based on Object Pascal, its benefits include easy-to-read code, quick compilation, and the use of multiple unit files for modular programming. Delphi has special features that support CodeGear's component framework and RAD environment. For the most part, descriptions and examples in this language guide assume that you are using CodeGear development tools.

Most developers using CodeGear software development tools write and compile their code in the integrated development environment (IDE). CodeGear development tools handle many details of setting up projects and source files, such as maintenance of dependency information among units. The product also places constraints on program organization that are not, strictly speaking, part of the Object Pascal language specification. For example, CodeGear development tools enforce certain file- and program-naming conventions that you can avoid if you write your programs outside of the IDE and compile them from the command prompt.

This language guide generally assumes that you are working in the IDE and that you are building applications that use the CodeGear Visual Component Library (VCL). Occasionally, however, Delphi-specific rules are distinguished from rules that apply to all Object Pascal programming. This text covers both the Win32 Delphi language compiler, and the Delphi for .NET language compiler. Platform-specific language differences and features are noted where necessary.

This section covers the following topics:

- Program Organization. Covers the basic language features that allow you to partition your application into units and namespaces.
- Example Programs. Small examples of both console and GUI applications are shown, with basic instructions on running the compiler from the command-line.

Program Organization

Delphi programs are usually divided into source-code modules called units. Most programs begin with a **program** heading, which specifies a name for the program. The **program** heading is followed by an optional **uses** clause, then a block of declarations and statements. The **uses** clause lists units that are linked into the program; these units, which can be shared by different programs, often have **uses** clauses of their own.

The **uses** clause provides the compiler with information about dependencies among modules. Because this information is stored in the modules themselves, most Delphi language programs do not require makefiles, header files, or preprocessor "include" directives.

Delphi Source Files

The compiler expects to find Delphi source code in files of three kinds:

- Unit source files (which end with the .pas extension)
- Project files (which end with the .dpr extension)
- Package source files (which end with the .dpk extension)

Unit source files typically contain most of the code in an application. Each application has a single project file and several unit files; the project file, which corresponds to the **program** file in traditional Pascal, organizes the unit files into an application. CodeGear development tools automatically maintain a project file for each application.

If you are compiling a program from the command line, you can put all your source code into unit (.pas) files. If you use the IDE to build your application, it will produce a project (.dpr) file.

Package source files are similar to project files, but they are used to construct special dynamically linkable libraries called packages.

Other Files Used to Build Applications

In addition to source-code modules, CodeGear products use several non-Pascal files to build applications. These files are maintained automatically by the IDE, and include

- VCL form files (which have a .dfm extension on Win32, and .nfm on .NET)
- Resource files (which end with .res)
- Project options files (which end with .dof)

A VCL form file contains the description of the properties of the form and the components it owns. Each form file represents a single form, which usually corresponds to a window or dialog box in an application. The IDE allows you to view and edit form files as text, and to save form files as either text (a format very suitable for version control) or binary. Although the default behavior is to save form files as text, they are usually not edited manually; it is more common to use CodeGear's visual design tools for this purpose. Each project has at least one form, and each form has an associated unit (.pas) file that, by default, has the same name as the form file.

In addition to VCL form files, each project uses a resource (.res) file to hold the application's icon and other resources such as strings. By default, this file has the same name as the project (.dpr) file.

A project options (.dof) file contains compiler and linker settings, search path information, version information, and so forth. Each project has an associated project options file with the same name as the project (.dpr) file. Usually, the options in this file are set from Project Options dialog.

Various tools in the IDE store data in files of other types. Desktop settings (.dsk) files contain information about the arrangement of windows and other configuration options; desktop settings can be project-specific or environment-wide. These files have no direct effect on compilation.

Compiler-Generated Files

The first time you build an application or a package, the compiler produces a compiled unit file (.dcu on Win32, .dcuil on .NET) for each new unit used in your project; all the .dcu/.dcuil files in your project are then linked to create a single executable or shared package. The first time you build a package, the compiler produces a file for each new unit contained in the package, and then creates both a .dcp and a package file.If you use the **GD** switch, the linker generates a map file and a .drc file; the .drc file, which contains string resources, can be compiled into a resource file.

When you build a project, individual units are not recompiled unless their source (.pas) files have changed since the last compilation, their .dcu/.dpu files cannot be found, you explicitly tell the compiler to reprocess them, or the interface of the unit depends on another unit which has been changed. In fact, it is not necessary for a unit's source file to be present at all, as long as the compiler can find the compiled unit file and that unit has no dependencies on other units that have changed.

Example Programs

The examples that follow illustrate basic features of Delphi programming. The examples show simple applications that would not normally be compiled from the IDE; you can compile them from the command line.

A Simple Console Application

The program below is a simple console application that you can compile and run from the command prompt.

```
program greeting;

  {$APPTYPE CONSOLE}
```

```
var MyMessage: string;

   begin
        MyMessage := 'Hello world!';
  Writeln(MyMessage);
end.
```

The first line declares a program called Greeting. The {$APPTYPE CONSOLE} directive tells the compiler that this is a console application, to be run from the command line. The next line declares a variable called MyMessage, which holds a string. (Delphi has genuine string data types.) The program then assigns the string "Hello world!" to the variable MyMessage, and sends the contents of MyMessage to the standard output using the Writeln procedure. (Writeln is defined implicitly in the System unit, which the compiler automatically includes in every application.)

You can type this program into a file called greeting.pas or greeting.dpr and compile it by entering

```
dcc32 greeting
```

to produce a Win32 executable, or

```
dccil greeting
```

to produce a managed .NET executable. In either case, the resulting executable prints the message Hello world!

Aside from its simplicity, this example differs in several important ways from programs that you are likely to write with CodeGear development tools. First, it is a console application. CodeGear development tools are most often used to write applications with graphical interfaces; hence, you would not ordinarily call Writeln. Moreover, the entire example program (save for Writeln) is in a single file. In a typical GUI application, the program heading the first line of the example would be placed in a separate project file that would not contain any of the actual application logic, other than a few calls to routines defined in unit files.

A More Complicated Example

The next example shows a program that is divided into two files: a project file and a unit file. The project file, which you can save as greeting.dpr, looks like this:

```
program greeting;

  {$APPTYPE CONSOLE}

     uses Unit1;

        begin
          PrintMessage('Hello World!');
  end.
```

The first line declares a program called greeting, which, once again, is a console application. The uses Unit1; clause tells the compiler that the program greeting depends on a unit called Unit1. Finally, the program calls the PrintMessage procedure, passing to it the string Hello World! The PrintMessage procedure is defined in Unit1. Here is the source code for Unit1, which must be saved in a file called Unit1.pas:

```
unit Unit1;

interface

procedure PrintMessage(msg: string);

implementation

procedure PrintMessage(msg: string);
begin
   Writeln(msg);
end;

end.
```

3

Unit1 defines a procedure called PrintMessage that takes a single string as an argument and sends the string to the standard output. (In Delphi, routines that do not return a value are called procedures. Routines that return a value are called functions.) Notice that PrintMessage is declared twice in Unit1. The first declaration, under the reserved word **interface**, makes PrintMessage available to other modules (such as greeting) that use Unit1. The second declaration, under the reserved word **implementation**, actually defines PrintMessage.

You can now compile Greeting from the command line by entering

```
dcc32 greeting
```

to produce a Win32 executable, or

```
dccil greeting
```

to produce a managed .NET executable.

There is no need to include Unit1 as a command-line argument. When the compiler processes greeting.dpr, it automatically looks for unit files that the greeting program depends on. The resulting executable does the same thing as our first example: it prints the message Hello world!

A VCL Application

Our next example is an application built using the Visual Component Library (VCL) components in the IDE. This program uses automatically generated form and resource files, so you won't be able to compile it from the source code alone. But it illustrates important features of the Delphi Language. In addition to multiple units, the program uses classes and objects (see page 514)

The program includes a project file and two new unit files. First, the project file:

```
program greeting;

        uses Forms, Unit1, Unit2;
        {$R *.res} // This directive links the project's resource file.

begin
   // Calls to global Application instance
   Application.Initialize;
   Application.CreateForm(TForm1, Form1);
   Application.CreateForm(TForm2, Form2);
   Application.Run;
end.
```

Once again, our program is called greeting. It uses three units: Forms, which is part of VCL; Unit1, which is associated with the application's main form (Form1); and Unit2, which is associated with another form (Form2).

The program makes a series of calls to an object named Application, which is an instance of the TApplication class defined in the Forms unit. (Every project has an automatically generated Application object.) Two of these calls invoke a TApplication method named CreateForm. The first call to CreateForm creates Form1, an instance of the TForm1 class defined in Unit1. The second call to CreateForm creates Form2, an instance of the TForm2 class defined in Unit2.

Unit1 looks like this:

```
unit Unit1;

interface

uses SysUtils, Types, Classes, Graphics, Controls, Forms, Dialogs;

type

TForm1 = class(TForm)
  Button1: TButton;
  procedure Button1Click(Sender: TObject);
end;
```

```
var
  Form1: TForm1;

implementation

uses Unit2;

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.ShowModal;
end;

end.
```

Unit1 creates a class named TForm1 (derived from TForm) and an instance of this class, Form1. TForm1 includes a buttonButton1, an instance of TButton and a procedure named Button1Click that is called when the user presses Button1. Button1Click hides Form1 and it displays Form2 (the call to Form2.ShowModal).

Note: In the previous example, Form2.ShowModal relies on the use of auto-created forms. While this is fine for example code, using auto-created forms is actively discouraged.

Form2 is defined in Unit2:

```
unit Unit2;

interface

uses SysUtils, Types, Classes, Graphics, Controls, Forms, Dialogs;

type
TForm2 = class(TForm)
  Label1: TLabel;
  CancelButton: TButton;
  procedure CancelButtonClick(Sender: TObject);
end;

var
  Form2: TForm2;

implementation

uses Unit1;

{$R *.dfm}

procedure TForm2.CancelButtonClick(Sender: TObject);
begin
   Form2.Close;
end;

end.
```

Unit2 creates a class named TForm2 and an instance of this class, Form2. TForm2 includes a button (CancelButton, an instance of TButton) and a label (Label1, an instance of TLabel). You can't see this from the source code, but Label1 displays a caption that reads Hello world! The caption is defined in Form2's form file, Unit2.dfm.

TForm2 declares and defines a method CancelButtonClick which will be invoked at runtime whenever the user presses CancelButton. This procedure (along with Unit1's TForm1.Button1Click) is called an *event handler* because it responds to events that occur while the program is running. Event handlers are assigned to specific events by the form files for Form1 and Form2.

When the greeting program starts, Form1 is displayed and Form2 is invisible. (By default, only the first form created in the project file is visible at runtime. This is called the project's main form.) When the user presses the button on Form1, Form2, displays the `Hello world!` greeting. When the user presses the `CancelButton` or the `Close` button on the title bar, `Form2` closes.

See Also

Programs and Units (↗ see page 683)

Using Namespaces with Delphi (↗ see page 689)

3.1.3.10 Procedures and Functions

This section describes the syntax of function and procedure declarations.

Topics

Name	Description
Procedures and Functions (↗ see page 662)	This topic covers the following items: • Declaring procedures and functions • Calling conventions • Forward and interface declarations • Declaration of external routines • Overloading procedures and functions • Local declarations and nested routines
Calling Procedures and Functions (↗ see page 669)	This topic covers the following items: • Program control and routine parameters • Open array constructors • The **inline** directive
Parameters (↗ see page 672)	This topic covers the following items: • Parameter semantics • String parameters • Array parameters • Default parameters

3.1.3.10.1 Procedures and Functions

This topic covers the following items:

- Declaring procedures and functions
- Calling conventions
- Forward and interface declarations
- Declaration of external routines
- Overloading procedures and functions
- Local declarations and nested routines

About Procedures and Functions

Procedures and functions, referred to collectively as *routines*, are self-contained statement blocks that can be called from different locations in a program. A function is a routine that returns a value when it executes. A procedure is a routine that does not return a value.

Function calls, because they return a value, can be used as expressions in assignments and operations. For example,

```
I := SomeFunction(X);
```

calls `SomeFunction` and assigns the result to `I`. Function calls cannot appear on the left side of an assignment statement.

Procedure calls - and, when extended syntax is enabled (`{$X+}`), function calls - can be used as complete statements. For example,

```
DoSomething;
```

calls the `DoSomething` routine; if `DoSomething` is a function, its return value is discarded.

Procedures and functions can call themselves recursively.

Declaring Procedures and Functions

When you declare a procedure or function, you specify its name, the number and type of parameters it takes, and, in the case of a function, the type of its return value; this part of the declaration is sometimes called the prototype, heading, or header. Then you write a block of code that executes whenever the procedure or function is called; this part is sometimes called the routine's body or block.

Procedure Declarations

A procedure declaration has the form

```
procedure procedureName(parameterList); directives;
  localDeclarations;
begin
  statements
end;
```

where *procedureName* is any valid identifier, *statements* is a sequence of statements that execute when the procedure is called, and *(parameterList)*, *directives;*, and *localDeclarations;* are optional.

Here is an example of a procedure declaration:

```
procedure NumString(N: Integer; var S: string);
var
  V: Integer;
begin
  V := Abs(N);
  S := '';
  repeat
    S := Chr(V mod 10 + Ord('0')) + S;
    V := V div 10;
  until V = 0;
  if N < 0 then S := '-' + S;
end;
```

Given this declaration, you can call the `NumString` procedure like this:

```
NumString(17, MyString);
```

This procedure call assigns the value '17' to `MyString` (which must be a **string** variable).

Within a procedure's statement block, you can use variables and other identifiers declared in the *localDeclarations* part of the procedure. You can also use the parameter names from the parameter list (like `N` and `S` in the previous example); the parameter list defines a set of local variables, so don't try to redeclare the parameter names in the *localDeclarations* section. Finally, you can use any identifiers within whose scope the procedure declaration falls.

Function Declarations

A function declaration is like a procedure declaration except that it specifies a return type and a return value. Function declarations have the form

```
function functionName(parameterList): returnType; directives;
  localDeclarations;
begin
  statements
end;
```

where *functionName* is any valid identifier, *returnType* is a type identifier, statements is a sequence of statements that execute when the function is called, and *(parameterList)*, *directives;*, and *localDeclarations;* are optional.

The function's statement block is governed by the same rules that apply to procedures. Within the statement block, you can use variables and other identifiers declared in the *localDeclarations* part of the function, parameter names from the parameter list, and any identifiers within whose scope the function declaration falls. In addition, the function name itself acts as a special variable that holds the function's return value, as does the predefined variable **Result**.

As long as extended syntax is enabled (`{$X+}`), **Result** is implicitly declared in every function. Do not try to redeclare it.

For example,

```
function WF: Integer;
begin
  WF := 17;
end;
```

defines a constant function called `WF` that takes no parameters and always returns the integer value 17. This declaration is equivalent to

```
function WF: Integer;
begin
  Result := 17;
end;
```

Here is a more complicated function declaration:

```
function Max(A: array of Real; N: Integer): Real;
var
X: Real;
I: Integer;
begin
  X := A[0];
  for I := 1 to N - 1 do
    if X < A[I] then X := A[I];
  Max := X;
end;
```

You can assign a value to **Result** or to the function name repeatedly within a statement block, as long as you assign only values that match the declared return type. When execution of the function terminates, whatever value was last assigned to **Result** or to the function name becomes the function's return value. For example,

```
function Power(X: Real; Y: Integer): Real;
var
  I: Integer;
begin
  Result := 1.0;
  I := Y;
  while I > 0 do
   begin
    if Odd(I) then Result := Result * X;
    I := I div 2;
    X := Sqr(X);
   end;
end;
```

Result and the function name always represent the same value. Hence

```
function MyFunction: Integer;
```

```
begin
  MyFunction := 5;
  Result := Result * 2;
  MyFunction := Result + 1;
end;
```

returns the value 11. But **Result** is not completely interchangeable with the function name. When the function name appears on the left side of an assignment statement, the compiler assumes that it is being used (like **Result**) to track the return value; when the function name appears anywhere else in the statement block, the compiler interprets it as a recursive call to the function itself. **Result**, on the other hand, can be used as a variable in operations, typecasts, set constructors, indexes, and calls to other routines.

If the function exits without assigning a value to **Result** or the function name, then the function's return value is undefined.

Calling Conventions

When you declare a procedure or function, you can specify a calling convention using one of the directives **register**, **pascal**, **cdecl**, **stdcall**, and **safecall**. For example,

```
function MyFunction(X, Y: Real): Real; cdecl;
```

Calling conventions determine the order in which parameters are passed to the routine. They also affect the removal of parameters from the stack, the use of registers for passing parameters, and error and exception handling. The default calling convention is **register**.

- The **register** and **pascal** conventions pass parameters from left to right; that is, the left most parameter is evaluated and passed first and the rightmost parameter is evaluated and passed last. The **cdecl**, **stdcall**, and **safecall** conventions pass parameters from right to left.
- For all conventions except **cdecl**, the procedure or function removes parameters from the stack upon returning. With the **cdecl** convention, the caller removes parameters from the stack when the call returns.
- The **register** convention uses up to three CPU registers to pass parameters, while the other conventions pass all parameters on the stack.
- The **safecall** convention implements exception 'firewalls.' On Win32, this implements interprocess COM error notification.

The table below summarizes calling conventions.

Calling conventions

Directive	Parameter order	Clean-up	Passes parameters in registers?
register	Left-to-right	Routine	Yes
pascal	Left-to-right	Routine	No
cdecl	Right-to-left	Caller	No
stdcall	Right-to-left	Routine	No
safecall	Right-to-left	Routine	No

The default **register** convention is the most efficient, since it usually avoids creation of a stack frame. (Access methods for published properties must use **register**.) The **cdecl** convention is useful when you call functions from shared libraries written in C or C++, while **stdcall** and **safecall** are recommended, in general, for calls to external code. On Win32, the operating system APIs are **stdcall** and **safecall**. Other operating systems generally use **cdecl**. (Note that **stdcall** is more efficient than **cdecl**.)

The **safecall** convention must be used for declaring dual-interface methods. The **pascal** convention is maintained for backward compatibility.

The directives **near**, **far**, and **export** refer to calling conventions in 16-bit Windows programming. They have no effect in Win32, or in .NET applications and are maintained for backward compatibility only.

Forward and Interface Declarations

The **forward** directive replaces the block, including local variable declarations and statements, in a procedure or function declaration. For example,

```
function Calculate(X, Y: Integer): Real; forward;
```

declares a function called `Calculate`. Somewhere after the **forward** declaration, the routine must be redeclared in a defining declaration that includes a block. The defining declaration for `Calculate` might look like this:

```
function Calculate;
  ... { declarations }
begin
  ... { statement block }
end;
```

Ordinarily, a defining declaration does not have to repeat the routine's parameter list or return type, but if it does repeat them, they must match those in the **forward** declaration exactly (except that default parameters can be omitted). If the **forward** declaration specifies an overloaded procedure or function, then the defining declaration must repeat the parameter list.

A **forward** declaration and its defining declaration must appear in the same **type** declaration section. That is, you can't add a new section (such as a **var** section or **const** section) between the forward declaration and the defining declaration. The defining declaration can be an **external** or **assembler** declaration, but it cannot be another **forward** declaration.

The purpose of a **forward** declaration is to extend the scope of a procedure or function identifier to an earlier point in the source code. This allows other procedures and functions to call the **forward**-declared routine before it is actually defined. Besides letting you organize your code more flexibly, **forward** declarations are sometimes necessary for mutual recursions.

The **forward** directive has no effect in the **interface** section of a unit. Procedure and function headers in the **interface** section behave like **forward** declarations and must have defining declarations in the **implementation** section. A routine declared in the **interface** section is available from anywhere else in the unit and from any other unit or program that uses the unit where it is declared.

External Declarations

The **external** directive, which replaces the block in a procedure or function declaration, allows you to call routines that are compiled separately from your program. External routines can come from object files or dynamically loadable libraries.

When importing a C function that takes a variable number of parameters, use the **varargs** directive. For example,

```
function printf(Format: PChar): Integer; cdecl; varargs;
```

The **varargs** directive works only with external routines and only with the **cdecl** calling convention.

Linking to Object Files

To call routines from a separately compiled object file, first link the object file to your application using the `$L` (or `$LINK`) compiler directive. For example,

```
{$L BLOCK.OBJ}
```

links `BLOCK.OBJ` into the program or unit in which it occurs. Next, declare the functions and procedures that you want to call:

```
procedure MoveWord(var Source, Dest; Count: Integer); external;
procedure FillWord(var Dest; Data: Integer; Count: Integer); external;
```

Now you can call the `MoveWord` and `FillWord` routines from `BLOCK.OBJ`.

On the Win32 platform, declarations like the ones above are frequently used to access external routines written in assembly language. You can also place assembly-language routines directly in your Delphi source code.

Importing Functions from Libraries

To import routines from a dynamically loadable library (.DLL), attach a directive of the form `external`*stringConstant*;

to the end of a normal procedure or function header, where *stringConstant* is the name of the library file in single quotation marks. For example, on Win32

```
function SomeFunction(S: string): string; external 'strlib.dll';
```

imports a function called `SomeFunction` from `strlib.dll`.

You can import a routine under a different name from the one it has in the library. If you do this, specify the original name in the **external** directive:

`external`*stringConstant1***name***stringConstant2;*

where the first *stringConstant* gives the name of the library file and the second *stringConstant* is the routine's original name.

The following declaration imports a function from `user32.dll` (part of the Win32 API).

```
function MessageBox(HWnd: Integer; Text, Caption: PChar; Flags: Integer): Integer; stdcall;
external 'user32.dll' name 'MessageBoxA';
```

The function's original name is `MessageBoxA`, but it is imported as `MessageBox`.

Instead of a name, you can use a number to identify the routine you want to import:

external*stringConstant***index***integerConstant;*

where *integerConstant* is the routine's index in the export table.

In your importing declaration, be sure to match the exact spelling and case of the routine's name. Later, when you call the imported routine, the name is case-insensitive.

Overloading Procedures and Functions

You can declare more than one routine in the same scope with the same name. This is called overloading. Overloaded routines must be declared with the **overload** directive and must have distinguishing parameter lists. For example, consider the declarations

```
function Divide(X, Y: Real): Real; overload;
begin
  Result := X/Y;
end

function Divide(X, Y: Integer): Integer; overload;
begin
  Result := X div Y;
end;
```

These declarations create two functions, both called `Divide`, that take parameters of different types. When you call `Divide`, the compiler determines which function to invoke by looking at the actual parameters passed in the call. For example, `Divide(6.0, 3.0)` calls the first `Divide` function, because its arguments are real-valued.

You can pass to an overloaded routine parameters that are not identical in type with those in any of the routine's declarations, but that are assignment-compatible with the parameters in more than one declaration. This happens most frequently when a routine is overloaded with different integer types or different real types - for example,

```
procedure Store(X: Longint); overload;
procedure Store(X: Shortint); overload;
```

In these cases, when it is possible to do so without ambiguity, the compiler invokes the routine whose parameters are of the type with the smallest range that accommodates the actual parameters in the call. (Remember that real-valued constant expressions are always of type **Extended**.)

Overloaded routines must be distinguished by the number of parameters they take or the types of their parameters. Hence the following pair of declarations causes a compilation error.

```
function Cap(S: string): string; overload;
  ...
procedure Cap(var Str: string); overload;
  ...
```

But the declarations

```
function Func(X: Real; Y: Integer): Real; overload;
  ...
function Func(X: Integer; Y: Real): Real; overload;
  ...
```

are legal.

When an overloaded routine is declared in a **forward** or interface declaration, the defining declaration must repeat the routine's parameter list.

The compiler can distinguish between overloaded functions that contain **AnsiString**/**PChar** and **WideString**/**WideChar** parameters in the same parameter position. String constants or literals passed into such an overload situation are translated into the native string or character type, which is **AnsiString**/**PChar**.

```
procedure test(const S: String);  overload;
procedure test(const W: WideString); overload;
var
  a: string;
  b: widestring;
begin
  a := 'a';
  b := 'b';
  test(a);    // calls String version
  test(b);    // calls WideString version
  test('abc');    // calls String version
  test(WideString('abc'));   // calls widestring version
end;
```

Variants can also be used as parameters in overloaded function declarations. Variant is considered more general than any simple type. Preference is always given to exact type matches over variant matches. If a variant is passed into such an overload situation, and an overload that takes a variant exists in that parameter position, it is considered to be an exact match for the Variant type.

This can cause some minor side effects with float types. Float types are matched by size. If there is no exact match for the float variable passed to the overload call but a variant parameter is available, the variant is taken over any smaller float type.

For example:

```
procedure foo(i: integer); overload;
procedure foo(d: double); overload;
procedure foo(v: variant); overload;
var
  v: variant;
begin
  foo(1);       // integer version
  foo(v);       // variant version
  foo(1.2);     // variant version (float literals -&gt; extended precision)
end;
```

This example calls the variant version of `foo`, not the double version, because the 1.2 constant is implicitly an extended type and extended is not an exact match for double. **Extended** is also not an exact match for **Variant**, but **Variant** is considered a more general type (whereas **double** is a smaller type than **extended**).

```
foo(Double(1.2));
```

This typecast does not work. You should use typed consts instead.

```
const  d: double = 1.2;
begin
```

```
   foo(d);
end;
```

The above code works correctly, and calls the double version.

```
const  s: single = 1.2;
begin
  foo(s);
end;
```

The above code also calls the double version of `foo`. **Single** is a better fit to double than to **variant**.

When declaring a set of overloaded routines, the best way to avoid float promotion to variant is to declare a version of your overloaded function for each float type (**Single**, **Double**, **Extended**) along with the **variant** version.

If you use default parameters in overloaded routines, be careful not to introduce ambiguous parameter signatures.

You can limit the potential effects of overloading by qualifying a routine's name when you call it. For example, `Unit1.MyProcedure(X, Y)` can call only routines declared in `Unit1;` if no routine in `Unit1` matches the name and parameter list in the call, an error results.

Local Declarations

The body of a function or procedure often begins with declarations of local variables used in the routine's statement block. These declarations can also include constants, types, and other routines. The scope of a local identifier is limited to the routine where it is declared.

Nested Routines

Functions and procedures sometimes contain other functions and procedures within the local-declarations section of their blocks. For example, the following declaration of a procedure called `DoSomething` contains a nested procedure.

```
procedure DoSomething(S: string);
var
  X, Y: Integer;

procedure NestedProc(S: string);
begin
  ...
end;

begin
  ...
  NestedProc(S);
  ...
end;
```

The scope of a nested routine is limited to the procedure or function in which it is declared. In our example, `NestedProc` can be called only within `DoSomething`.

For real examples of nested routines, look at the DateTimeToString procedure, the **ScanDate** function, and other routines in the `SysUtils` unit.

See Also

Parameters (see page 672)

Calling Procedures and Functions (see page 669)

3.1.3.10.2 Calling Procedures and Functions

This topic covers the following items:

- Program control and routine parameters

- Open array constructors
- The **inline** directive

Program Control and Parameters

When you call a procedure or function, program control passes from the point where the call is made to the body of the routine. You can make the call using the routine's declared name (with or without qualifiers) or using a procedural variable that points to the routine. In either case, if the routine is declared with parameters, your call to it must pass parameters that correspond in order and type to the routine's parameter list. The parameters you pass to a routine are called actual parameters, while the parameters in the routine's declaration are called formal parameters.

When calling a routine, remember that

- expressions used to pass typed **const** and value parameters must be assignment-compatible with the corresponding formal parameters.
- expressions used to pass **var** and **out** parameters must be identically typed with the corresponding formal parameters, unless the formal parameters are untyped.
- only assignable expressions can be used to pass **var** and **out** parameters.
- if a routine's formal parameters are untyped, numerals and true constants with numeric values cannot be used as actual parameters.

When you call a routine that uses default parameter values, all actual parameters following the first accepted default must also use the default values; calls of the form `SomeFunction(,,X)` are not legal.

You can omit parentheses when passing all and only the default parameters to a routine. For example, given the procedure

```
procedure DoSomething(X: Real = 1.0; I: Integer = 0; S: string = '');
```

the following calls are equivalent.

```
DoSomething();
          DoSomething;
```

Open Array Constructors

Open array constructors allow you to construct arrays directly within function and procedure calls. They can be passed only as open array parameters or variant open array parameters.

An open array constructor, like a set constructor, is a sequence of expressions separated by commas and enclosed in brackets.

For example, given the declarations

```
var I, J: Integer;
          procedure Add(A: array of Integer);
```

you could call the Add procedure with the statement

```
Add([5, 7, I, I + J]);
```

This is equivalent to

```
var Temp: array[0..3] of Integer;
            ...
          Temp[0] := 5;
          Temp[1] := 7;
          Temp[2] := I;
          Temp[3] := I + J;
          Add(Temp);
```

Open array constructors can be passed only as value or **const** parameters. The expressions in a constructor must be assignment-compatible with the base type of the array parameter. In the case of a variant open array parameter, the expressions can be of different types.

Using the inline Directive

The Delphi compiler allows functions and procedures to be tagged with the

inline directive to improve performance. If the function or procedure meets certain criteria, the compiler will insert code directly, rather than generating a call. Inlining is a performance optimization that can result in faster code, but at the expense of space. Inlining always causes the compiler to produce a larger binary file. The **inline** directive is used in function and procedure declarations and definitions, like other directives.

```
procedure MyProc(x:Integer); inline;
begin
    // ...
end;

function MyFunc(y:Char) : String; inline;
begin
   // ..
end;
```

The **inline** directive is a suggestion to the compiler. There is no guarantee the compiler will inline a particular routine, as there are a number of circumstances where inlining cannot be done. The following list shows the conditions under which inlining does or does not occur:

- Inlining will not occur on any form of late-bound method. This includes virtual, dynamic, and message methods.
- Routines containing assembly code will not be inlined.
- Constructors and destructors will not be inlined.
- The main program block, unit initialization, and unit finalization blocks cannot be inlined.
- Routines that are not defined before use cannot be inlined.
- Routines that take open array parameters cannot be inlined.
- Code can be inlined within packages, however, inlining never occurs across package boundaries.
- No inlining will be done between units that are circularly dependent. This included indirect circular dependencies, for example, unit A uses unit B, and unit B uses unit C which in turn uses unit A. In this example, when compiling unit A, no code from unit B or unit C will be inlined in unit A.
- The compiler can inline code when a unit is in a circular dependency, as long as the code to be inlined comes from a unit outside the circular relationship. In the above example, if unit A also used unit D, code from unit D could be inlined in A, since it is not involved in the circular dependency.
- If a routine is defined in the **interface** section and it accesses symbols defined in the **implementation** section, that routine cannot be inlined.
- In Delphi.NET, routines in classes cannot be inlined if they access members with less (i.e. more restricted) visibility than the method itself. For example, if a public method accesses private symbols, it cannot be inlined.
- If a routine marked with **inline** uses external symbols from other units, all of those units must be listed in the **uses** statement, otherwise the routine cannot be inlined.
- Procedures and functions used in conditional expressions in **while**-**do** and **repeat**-**until** statements cannot be expanded inline.
- Within a unit, the body for an inline function should be defined before calls to the function are made. Otherwise, the body of the function, which is not known to the compiler when it reaches the call site, cannot be expanded inline.

If you modify the implementation of an inlined routine, you will cause all units that use that function to be recompiled. This is different from traditional rebuild rules, where rebuilds were triggered only by changes in the **interface** section of a unit.

The `{$INLINE}` compiler directive gives you finer control over inlining. The `{$INLINE}` directive can be used at the site of the inlined routine's definition, as well as at the call site. Below are the possible values and their meaning:

Value	Meaning at definition	Meaning at call site
`{$INLINE ON}` (default)	The routine is compiled as inlineable if it is tagged with the **inline** directive.	The routine will be expanded inline if possible.

{$INLINE AUTO}	Behaves like {$INLINE ON}, with the addition that routines not marked with **inline** will be inlined if their code size is less than or equal to 32 bytes.	{$INLINE AUTO} has no effect on whether a routine will be inlined when it is used at the call site of the routine.
{$INLINE OFF}	The routine will not be marked as inlineable, even if it is tagged with inline.	The routine will not be expanded inline.

See Also

Procedures and Functions (see page 662)

Parameters (see page 672)

3.1.3.10.3 Parameters

This topic covers the following items:

- Parameter semantics
- String parameters
- Array parameters
- Default parameters

About Parameters

Most procedure and function headers include a parameter list. For example, in the header

```
function Power(X: Real; Y: Integer): Real;
```

the parameter list is `(X: Real; Y: Integer)`.

A parameter list is a sequence of parameter declarations separated by semicolons and enclosed in parentheses. Each declaration is a comma-delimited series of parameter names, followed in most cases by a colon and a type identifier, and in some cases by the **=** symbol and a default value. Parameter names must be valid identifiers. Any declaration can be preceded by **var**, **const**, or **out**. Examples:

```
(X, Y: Real)
(var S: string; X: Integer)
(HWnd: Integer; Text, Caption: PChar; Flags: Integer)
(const P; I: Integer)
```

The parameter list specifies the number, order, and type of parameters that must be passed to the routine when it is called. If a routine does not take any parameters, omit the identifier list and the parentheses in its declaration:

```
procedure UpdateRecords;
begin
  ...
end;
```

Within the procedure or function body, the parameter names (X and Y in the first example) can be used as local variables. Do not redeclare the parameter names in the local declarations section of the procedure or function body.

Parameter Semantics

Parameters are categorized in several ways:

- Every parameter is classified as value, variable, constant, or out. Value parameters are the default; the reserved words **var**, **const**, and **out** indicate variable, constant, and out parameters, respectively.
- Value parameters are always typed, while constant, variable, and out parameters can be either typed or untyped.
- Special rules apply to array parameters.

Files and instances of structured types that contain files can be passed only as variable (**var**) parameters.

Value and Variable Parameters

Most parameters are either value parameters (the default) or variable (**var**) parameters. Value parameters are passed by value, while variable parameters are passed by reference. To see what this means, consider the following functions.

```
function DoubleByValue(X: Integer): Integer;   // X is a value parameter
begin
  X := X * 2;
  Result := X;
end;

function DoubleByRef(var X: Integer): Integer;  // X is a variable parameter
begin
  X := X * 2;
  Result := X;
end;
```

These functions return the same result, but only the second one - `DoubleByRefcan` change the value of a variable passed to it. Suppose we call the functions like this:

```
var
  I, J, V, W: Integer;
begin
  I := 4;
  V := 4;
  J := DoubleByValue(I);   // J = 8, I = 4
  W := DoubleByRef(V);     // W = 8, V = 8
end;
```

After this code executes, the variable `I`, which was passed to `DoubleByValue`, has the same value we initially assigned to it. But the variable V, which was passed to `DoubleByRef`, has a different value.

A value parameter acts like a local variable that gets initialized to the value passed in the procedure or function call. If you pass a variable as a value parameter, the procedure or function creates a copy of it; changes made to the copy have no effect on the original variable and are lost when program execution returns to the caller.

A variable parameter, on the other hand, acts like a pointer rather than a copy. Changes made to the parameter within the body of a function or procedure persist after program execution returns to the caller and the parameter name itself has gone out of scope.

Even if the same variable is passed in two or more **var** parameters, no copies are made. This is illustrated in the following example.

```
procedure AddOne(var X, Y: Integer);
begin
  X := X + 1;
  Y := Y + 1;
end;

var I: Integer;
begin
  I := 1;
  AddOne(I, I);
end;
```

After this code executes, the value of `I` is 3.

If a routine's declaration specifies a **var** parameter, you must pass an assignable expression - that is, a variable, typed constant (in the `{$J+}` state), dereferenced pointer, field, or indexed variable to the routine when you call it. To use our previous examples, `DoubleByRef(7)` produces an error, although `DoubleByValue(7)` is legal.

Indexes and pointer dereferences passed in **var** parameters - for example, `DoubleByRef(MyArray[I])` - are evaluated once, before execution of the routine.

Constant Parameters

A constant (**const**) parameter is like a local constant or read-only variable. Constant parameters are similar to value parameters, except that you can't assign a value to a constant parameter within the body of a procedure or function, nor can you pass one as a **var** parameter to another routine. (But when you pass an object reference as a constant parameter, you can still modify the object's properties.)

Using **const** allows the compiler to optimize code for structured - and string-type parameters. It also provides a safeguard against unintentionally passing a parameter by reference to another routine.

Here, for example, is the header for the `CompareStr` function in the `SysUtils` unit:

```
function CompareStr(const S1, S2: string): Integer;
```

Because `S1` and `S2` are not modified in the body of CompareStr, they can be declared as constant parameters.

Out Parameters

An **out** parameter, like a variable parameter, is passed by reference. With an **out** parameter, however, the initial value of the referenced variable is discarded by the routine it is passed to. The **out** parameter is for output only; that is, it tells the function or procedure where to store output, but doesn't provide any input.

For example, consider the procedure heading

```
procedure GetInfo(out Info: SomeRecordType);
```

When you call `GetInfo`, you must pass it a variable of type `SomeRecordType`:

```
var MyRecord: SomeRecordType;
   ...
GetInfo(MyRecord);
```

But you're not using `MyRecord` to pass any data to the `GetInfo` procedure; `MyRecord` is just a container where you want `GetInfo` to store the information it generates. The call to `GetInfo` immediately frees the memory used by `MyRecord`, before program control passes to the procedure.

Out parameters are frequently used with distributed-object models like COM. In addition, you should use **out** parameters when you pass an uninitialized variable to a function or procedure.

Untyped Parameters

You can omit type specifications when declaring **var**, **const**, and **out** parameters. (Value parameters must be typed.) For example,

```
procedure TakeAnything(const C);
```

declares a procedure called `TakeAnything` that accepts a parameter of any type. When you call such a routine, you cannot pass it a numeral or untyped numeric constant.

Within a procedure or function body, untyped parameters are incompatible with every type. To operate on an untyped parameter, you must cast it. In general, the compiler cannot verify that operations on untyped parameters are valid.

The following example uses untyped parameters in a function called `Equal` that compares a specified number of bytes of any two variables.

```
function Equal(var Source, Dest; Size: Integer): Boolean;
type
  TBytes = array[0..MaxInt - 1] of Byte;
var
  N : Integer;
begin
  N := 0;
  while (N < Size) and (TBytes(Dest)[N] = TBytes(Source)[N]) do
```

```
    Inc(N);
    Equal := N = Size;
end;
```

Given the declarations

```
type
  TVector = array[1..10] of Integer;
  TPoint = record
    X, Y: Integer;
  end;
var
  Vec1, Vec2: TVector;
  N: Integer;
  P: TPoint;
```

you could make the following calls to `Equal`:

```
Equal(Vec1, Vec2, SizeOf(TVector));      // compare Vec1 to Vec2
Equal(Vec1, Vec2, SizeOf(Integer) * N);  // compare first N elements of Vec1 and Vec2
Equal(Vec1[1], Vec1[6], SizeOf(Integer) * 5);   // compare first 5 to last 5 elements of Vec1
Equal(Vec1[1], P, 4);                    // compare Vec1[1] to P.X and Vec1[2] to P.Y
```

String Parameters

When you declare routines that take short-string parameters, you cannot include length specifiers in the parameter declarations. That is, the declaration

```
procedure Check(S: string[20]);   // syntax error
```

causes a compilation error. But

```
type TString20 = string[20];
procedure Check(S: TString20);
```

is valid. The special identifier **OpenString** can be used to declare routines that take short-string parameters of varying length:

```
procedure Check(S: OpenString);
```

When the `{$H}` and `{$P+}` compiler directives are both in effect, the reserved word **string** is equivalent to **OpenString** in parameter declarations.

Short strings, **OpenString**, `$H`, and `$P` are supported for backward compatibility only. In new code, you can avoid these considerations by using long strings.

Array Parameters

When you declare routines that take array parameters, you cannot include index type specifiers in the parameter declarations. That is, the declaration

```
procedure Sort(A: array[1..10] of Integer)  // syntax error<
```

causes a compilation error. But

```
type TDigits = array[1..10] of Integer;
procedure Sort(A: TDigits);
```

is valid. Another approach is to use open array parameters.

Since the Delphi language does not implement value semantics for dynamic arrays, 'value' parameters in routines do not represent a full copy of the dynamic array. In this example

```
type
  TDynamicArray = array of Integer;
  procedure p(Value: TDynamicArray);
  begin
    Value[0] := 1;
  end;
  procedure Run;
var
    a: TDynamicArray;
begin
  SetLength(a, 1);
  a[0] := 0;
  p(a);
  Writeln(a[0]); // Prints '1'
end;
```

Note that the assignment to `Value[0]` in routine `p` will modify the content of dynamic array of the caller, despite `Value` being a by-value parameter. If a full copy of the dynamic array is required, use the Copy standard procedure to create a value copy of the dynamic array.

Open Array Parameters

Open array parameters allow arrays of different sizes to be passed to the same procedure or function. To define a routine with an open array parameter, use the syntax `array of` *type* (rather than `array[X..Y] of` *type*) in the parameter declaration. For example,

```
function Find(A: array of Char): Integer;
```

declares a function called `Find` that takes a character array of any size and returns an integer.

Note: The syntax of open array parameters resembles that of dynamic array types, but they do not mean the same thing. The previous example creates a function that takes any array of Char

elements, including (but not limited to) dynamic arrays. To declare parameters that must be dynamic arrays, you need to specify a type identifier:

```
type TDynamicCharArray = array of Char;
function Find(A: TDynamicCharArray): Integer;
```

Within the body of a routine, open array parameters are governed by the following rules.

- They are always zero-based. The first element is 0, the second element is 1, and so forth. The standard **Low** and **High** functions return 0 and Length1, respectively. The **SizeOf** function returns the size of the actual array passed to the routine.
- They can be accessed by element only. Assignments to an entire open array parameter are not allowed.
- They can be passed to other procedures and functions only as open array parameters or untyped **var** parameters. They cannot be passed to SetLength.
- Instead of an array, you can pass a variable of the open array parameter's base type. It will be treated as an array of length 1.

When you pass an array as an open array value parameter, the compiler creates a local copy of the array within the routine's stack frame. Be careful not to overflow the stack by passing large arrays.

The following examples use open array parameters to define a `Clear` procedure that assigns zero to each element in an array of reals and a `Sum` function that computes the sum of the elements in an array of reals.

```
procedure Clear(var A: array of Real);
var
   I: Integer;
begin
   for I := 0 to High(A) do A[I] := 0;
end;

function Sum(const A: array of Real): Real;
var
  I: Integer;
  S: Real;
begin
  S := 0;
  for I := 0 to High(A) do S := S + A[I];
  Sum := S;
end;
```

When you call routines that use open array parameters, you can pass open array constructors to them.

Variant Open Array Parameters

Variant open array parameters allow you to pass an array of differently typed expressions to a single procedure or function. To define a routine with a variant open array parameter, specify `array of const` as the parameter's type. Thus

```
procedure DoSomething(A: array of const);
```

declares a procedure called `DoSomething` that can operate on heterogeneous arrays.

On the .NET platform, a variant open array parameter is equivalent to `array of TObject`. To determine the type of an element in the array, you may use the ClassName or GetType methods.

On the Win32 platform, the `array of const` construction is equivalent to `array ofTVarRec`. `TVarRec`, declared in the `System` unit, represents a record with a variant part that can hold values of integer, Boolean, character, real, string, pointer, class, class reference, interface, and variant types. `TVarRec`'s `VType` field indicates the type of each element in the array. Some types are passed as pointers rather than values; in particular, long strings are passed as **Pointer** and must be typecast to **string**.

The following Win32 example, uses a variant open array parameter in a function that creates a string representation of each element passed to it and concatenates the results into a single string. The string-handling routines called in this function are defined in `SysUtils`. This function will not compile on .NET because it depends on the variant implementation of `array of const`.

```
function MakeStr(const Args: array of const): string;
var
  I: Integer;
begin
  Result := '';
  for I := 0 to High(Args) do
     with Args[I] do
        case VType of
            vtInteger:  Result := Result + IntToStr(VInteger);
            vtBoolean:  Result := Result + BoolToStr(VBoolean);
            vtChar:     Result := Result + VChar;
            vtExtended: Result := Result + FloatToStr(VExtended^);
            vtString:   Result := Result + VString^;
            vtPChar:    Result := Result + VPChar;
            vtObject:   Result := Result + VObject.ClassName;
            vtClass:    Result := Result + VClass.ClassName;
            vtAnsiString:  Result := Result + string(VAnsiString);
            vtCurrency:    Result := Result + CurrToStr(VCurrency^);
            vtVariant:     Result := Result + string(VVariant^);
            vtInt64:       Result := Result + IntToStr(VInt64^);
  end;
end;
```

We can call this function using an open array constructor. For example,

```
MakeStr(['test', 100, ' ', True, 3.14159, TForm])
```

returns the string `'test100 T3.14159TForm'`.

Default Parameters

You can specify default parameter values in a procedure or function heading. Default values are allowed only for typed **const** and value parameters. To provide a default value, end the parameter declaration with the **=** symbol followed by a constant expression that is assignment-compatible with the parameter's type.

For example, given the declaration

```
procedure FillArray(A: array of Integer; Value: Integer = 0);
```

the following procedure calls are equivalent.

```
FillArray(MyArray);
```

```
          FillArray(MyArray, 0);
```

A multiple-parameter declaration cannot specify a default value. Thus, while

```
function MyFunction(X: Real = 3.5; Y: Real = 3.5): Real;
```

is legal,

```
function MyFunction(X, Y: Real = 3.5): Real;  // syntax error
```

is not.

Parameters with default values must occur at the end of the parameter list. That is, all parameters following the first declared default value must also have default values. So the following declaration is illegal.

```
procedure MyProcedure(I: Integer = 1; S: string);  // syntax error
```

Default values specified in a procedural type override those specified in an actual routine. Thus, given the declarations

```
type TResizer = function(X: Real; Y: Real = 1.0): Real;
function Resizer(X: Real; Y: Real = 2.0): Real;
var
  F: TResizer;
  N: Real;
```

the statements

```
F := Resizer;
F(N);
```

result in the values `(N, 1.0)` being passed to `Resizer`.

Default parameters are limited to values that can be specified by a constant expression. Hence parameters of a dynamic-array, procedural, class, class-reference, or interface type can have no value other than **nil** as their default. Parameters of a record, variant, file, static-array, or object type cannot have default values at all.

Default Parameters and Overloaded Functions

If you use default parameter values in an overloaded routine, avoid ambiguous parameter signatures. Consider, for example, the following.

```
procedure Confused(I: Integer); overload;
   ...
procedure Confused(I: Integer; J: Integer = 0); overload;
   ...
Confused(X);   //  Which procedure is called?
```

In fact, neither procedure is called. This code generates a compilation error.

Default Parameters in Forward and Interface Declarations

If a routine has a **forward** declaration or appears in the interface section of a unit, default parameter values if there are any must be specified in the **forward** or interface declaration. In this case, the default values can be omitted from the defining (implementation) declaration; but if the defining declaration includes default values, they must match those in the **forward** or interface declaration exactly.

See Also

Procedures and Functions (see page 662)

Calling Procedures and Functions (see page 669)

3.1.3.11 Program Control

This section describes how parameters are passed to procedures and functions.

Topics

Name	Description
Program Control (↗ see page 679)	The concepts of passing parameters and function result processing are important to understand before you undertake your application projects Treatment of parameters and function results is determined by several factors, including calling conventions, parameter semantics, and the type and size of the value being passed. This following topics are covered in this material: • Passing Parameters. • Handling Function Results. • Handling Method Calls. • Understanding Exit Procedures.

3.1.3.11.1 Program Control

The concepts of passing parameters and function result processing are important to understand before you undertake your application projects Treatment of parameters and function results is determined by several factors, including calling conventions, parameter semantics, and the type and size of the value being passed.

This following topics are covered in this material:

- Passing Parameters.
- Handling Function Results.
- Handling Method Calls.
- Understanding Exit Procedures.

Passing Parameters

Parameters are transferred to procedures and functions via CPU registers or the stack, depending on the routine's calling convention. For information about calling conventions, see the topic on Calling Conventions.

By Value vs. By Reference

Variable (**var**) parameters are always passed by reference, as 32-bit pointers that point to the actual storage location.

Value and constant (**const**) parameters are passed by value or by reference, depending on the type and size of the parameter:

- An ordinal parameter is passed as an 8-bit, 16-bit, 32-bit, or 64-bit value, using the same format as a variable of the corresponding type.
- A real parameter is always passed on the stack. A Single parameter occupies 4 bytes, and a Double, Comp, or Currency parameter occupies 8 bytes. A Real48 occupies 8 bytes, with the Real48 value stored in the lower 6 bytes. An Extended occupies 12 bytes, with the Extended value stored in the lower 10 bytes.
- A short-string parameter is passed as a 32-bit pointer to a short string.
- A long-string or dynamic-array parameter is passed as a 32-bit pointer to the dynamic memory block allocated for the long string. The value **nil** is passed for an empty long string.
- A pointer, class, class-reference, or procedure-pointer parameter is passed as a 32-bit pointer.
- A method pointer is passed on the stack as two 32-bit pointers. The instance pointer is pushed before the method pointer so that the method pointer occupies the lowest address.
- Under the **register** and **pascal** conventions, a variant parameter is passed as a 32bit pointer to a Variant value.
- Sets, records, and static arrays of 1, 2, or 4 bytes are passed as 8-bit, 16-bit, and 32bit values. Larger sets, records, and static arrays are passed as 32-bit pointers to the value. An exception to this rule is that records are always passed directly on the stack under the **cdecl**, **stdcall**, and **safecall** conventions; the size of a record passed this way is rounded upward to the nearest double-word boundary.

- An open-array parameter is passed as two 32-bit values. The first value is a pointer to the array data, and the second value is one less than the number of elements in the array.

When two parameters are passed on the stack, each parameter occupies a multiple of 4 bytes (a whole number of double words). For an 8-bit or 16-bit parameter, even though the parameter occupies only a byte or a word, it is passed as a double word. The contents of the unused parts of the double word are undefined.

Pascal, cdecl, stdcall, and safecall Conventions

Under the **pascal**, **cdecl**, **stdcall** and **safecall** conventions, all parameters are passed on the stack. Under the **pascal** convention, parameters are pushed in the order of their declaration (left-to-right), so that the first parameter ends up at the highest address and the last parameter ends up at the lowest address. Under the **cdecl**, **stdcall**, and **safecall** conventions, parameters are pushed in reverse order of declaration (right-to-left), so that the first parameter ends up at the lowest address and the last parameter ends up at the highest address.

Register Convention

Under the **register** convention, up to three parameters are passed in CPU registers, and the rest (if any) are passed on the stack. The parameters are passed in order of declaration (as with the **pascal** convention), and the first three parameters that qualify are passed in the EAX, EDX, and ECX registers, in that order. Real, method-pointer, variant, Int64, and structured types do not qualify as register parameters, but all other parameters do. If more than three parameters qualify as register parameters, the first three are passed in EAX, EDX, and ECX, and the remaining parameters are pushed onto the stack in order of declaration. For example, given the declaration

```
procedure Test(A: Integer; var B: Char; C: Double; const D: string; E: Pointer);
```

a call to Test passes A in EAX as a 32-bit integer, B in EDX as a pointer to a Char, and D in ECX as a pointer to a long-string memory block; C and E are pushed onto the stack as two double-words and a 32-bit pointer, in that order.

Register saving conventions

Procedures and functions must preserve the EBX, ESI, EDI, and EBP registers, but can modify the EAX, EDX, and ECX registers. When implementing a constructor or destructor in assembler, be sure to preserve the DL register. Procedures and functions are invoked with the assumption that the CPU's direction flag is cleared (corresponding to a CLD instruction) and must return with the direction flag cleared.

Note: Delphi language procedures and functions are generally invoked with the assumption that the FPU stack is empty: The compiler tries to use all eight FPU stack entries when it generates code.

When working with the MMX and XMM instructions, be sure to preserve the values of the xmm and mm registers. Delphi functions are invoked with the assumption that the x87 FPU data registers are available for use by x87 floating point instructions. That is, the compiler assumes that the EMMS/FEMMS instruction has been called after MMX operations. Delphi functions do not make any assumptions about the state and content of xmm registers. They do not guarantee that the content of xmm registers is unchanged.

Handling Function Results

The following conventions are used for returning function result values.

- Ordinal results are returned, when possible, in a CPU register. Bytes are returned in AL, words are returned in AX, and double-words are returned in EAX.
- Real results are returned in the floating-point coprocessor's top-of-stack register (ST(0)). For function results of type Currency, the value in ST(0) is scaled by 10000. For example, the Currency value 1.234 is returned in ST(0) as 12340.
- For a string, dynamic array, method pointer, or variant result, the effects are the same as if the function result were declared as an additional **var** parameter following the declared parameters. In other words, the caller passes an additional 32-bit pointer that points to a variable in which to return the function result.
- Int64 is returned in EDX:EAX.
- Pointer, class, class-reference, and procedure-pointer results are returned in EAX.

- For static-array, record, and set results, if the value occupies one byte it is returned in AL; if the value occupies two bytes it is returned in AX; and if the value occupies four bytes it is returned in EAX. Otherwise, the result is returned in an additional **var** parameter that is passed to the function after the declared parameters.

Handling Method Calls

Methods use the same calling conventions as ordinary procedures and functions, except that every method has an additional implicit parameter Self, which is a reference to the instance or class in which the method is called. The Self parameter is passed as a 32-bit pointer.

- Under the **register** convention, Self behaves as if it were declared before all other parameters. It is therefore always passed in the EAX register.
- Under the **pascal** convention, Self behaves as if it were declared after all other parameters (including the additional **var** parameter sometimes passed for a function result). It is therefore pushed last, ending up at a lower address than all other parameters.
- Under the **cdecl**, **stdcall**, and **safecall** conventions, Self behaves as if it were declared before all other parameters, but after the additional **var** parameter (if any) passed for a function result. It is therefore the last to be pushed, except for the additional **var** parameter.

Constructors and destructors use the same calling conventions as other methods, except that an additional Boolean flag parameter is passed to indicate the context of the constructor or destructor call.

A value of False in the flag parameter of a constructor call indicates that the constructor was invoked through an instance object or using the **inherited** keyword. In this case, the constructor behaves like an ordinary method. A value of True in the flag parameter of a constructor call indicates that the constructor was invoked through a class reference. In this case, the constructor creates an instance of the class given by Self, and returns a reference to the newly created object in EAX.

A value of False in the flag parameter of a destructor call indicates that the destructor was invoked using the **inherited** keyword. In this case, the destructor behaves like an ordinary method. A value of True in the flag parameter of a destructor call indicates that the destructor was invoked through an instance object. In this case, the destructor deallocates the instance given by Self just before returning.

The flag parameter behaves as if it were declared before all other parameters. Under the **register** convention, it is passed in the DL register. Under the **pascal** convention, it is pushed before all other parameters. Under the **cdecl**, **stdcall**, and **safecall** conventions, it is pushed just before the Self parameter.

Since the DL register indicates whether the constructor or destructor is the outermost in the call stack, you must restore the value of DL before exiting so that BeforeDestruction or AfterConstruction can be called properly.

Understanding Exit Procedures

Exit procedures ensure that specific actions such as updating and closing filesare carried out before a program terminates. The `ExitProc` pointer variable allows you to *install* an exit procedure, so that it is always called as part of the program's termination whether the termination is normal, forced by a call to Halt, or the result of a runtime error. An exit procedure takes no parameters.

Note: It is recommended that you use finalization sections rather than exit procedures for all exit behavior. `Exit` procedures are available only for executables. For .DLLs (Win32) you can use a similar variable, *DllProc*, which is called when the library is loaded as well as when it is unloaded. For packages, exit behavior must be implemented in a finalization section. All exit procedures are called before execution of finalization sections.

Units as well as programs can install exit procedures. A unit can install an exit procedure as part of its initialization code, relying on the procedure to close files or perform other clean-up tasks.

When implemented properly, an exit procedure is part of a chain of exit procedures. The procedures are executed in reverse order of installation, ensuring that the exit code of one unit isn't executed before the exit code of any units that depend on it. To keep the chain intact, you must save the current contents of `ExitProc` before pointing it to the address of your own exit procedure. Also, the first statement in your exit procedure must reinstall the saved value of `ExitProc`.

The following code shows a skeleton implementation of an exit procedure.

```
var
```

```
ExitSave: Pointer;

procedure MyExit;

begin
    ExitProc := ExitSave; // always restore old vector first
 .
    .
    .
end;

begin
    ExitSave := ExitProc;
    ExitProc := @MyExit;
    .
    .
    .
end.
```

On entry, the code saves the contents of `ExitProc` in `ExitSave`, then installs the `MyExit` procedure. When called as part of the termination process, the first thing `MyExit` does is reinstall the previous exit procedure.

The termination routine in the runtime library keeps calling exit procedures until `ExitProc` becomes **nilnil**. To avoid infinite loops, `ExitProc` is set to **nil** before every call, so the next exit procedure is called only if the current exit procedure assigns an address to `ExitProc`. If an error occurs in an exit procedure, it is not called again.

An exit procedure can learn the cause of termination by examining the `ExitCode` integer variable and the `ErrorAddr` pointer variable. In case of normal termination, `ExitCode` is zero and `ErrorAddr` is **nil**. In case of termination through a call to `Halt`, `ExitCode` contains the value passed to Halt and ErrorAddr is **nil**. In case of termination due to a runtime error, ExitCode contains the error code and ErrorAddr contains the address of the invalid statement.

The last exit procedure (the one installed by the runtime library) closes the Input and Output files. If ErrorAddr is not **nil**, it outputs a runtime error message. To output your own runtime error message, install an exit procedure that examines ErrorAddr and outputs a message if it's not **nil**; before returning, set ErrorAddr to **nil** so that the error is not reported again by other exit procedures.

Once the runtime library has called all exit procedures, it returns to the operating system, passing the value stored in ExitCode as a return code.

See Also

Procedures and Functions (⧉ see page 662)

3.1.3.12 Programs and Units

This chapter provides a more detailed look at Delphi program organization.

Topics

Name	Description
Programs and Units (↗ see page 683)	A Delphi program is constructed from source code modules called units. The units are tied together by a special source code module that contains either the **program**, **library**, or **package** header. Each unit is stored in its own file and compiled separately; compiled units are linked to create an application. RAD Studio introduces hierarchical namespaces, giving you even more flexibility in organizing your units. Namespaces and units allow you to • Divide large programs into modules that can be edited separately. • Create libraries that you can share among programs. • Distribute libraries to other developers without making the source code... more (↗ see page 683)
Using Namespaces with Delphi (↗ see page 689)	In Delphi, a unit is the basic container for types. Microsoft's Common Language Runtime (CLR) introduces another layer of organization called a namespace. In the .NET Framework, a namespace is a conceptual container of types. In Delphi, a namespace is a container of Delphi units. The addition of namespaces gives Delphi the ability to access and extend classes in the .NET Framework. Unlike traditional Delphi units, namespaces can be nested to form a containment hierarchy. Nested namespaces provide a way to organize identifiers and types, and are used to disambiguate types with the same name. Since they are a container... more (↗ see page 689)

3.1.3.12.1 Programs and Units

A Delphi program is constructed from source code modules called units. The units are tied together by a special source code module that contains either the **program**, **library**, or **package** header. Each unit is stored in its own file and compiled separately; compiled units are linked to create an application. RAD Studio introduces hierarchical namespaces, giving you even more flexibility in organizing your units. Namespaces and units allow you to

- Divide large programs into modules that can be edited separately.
- Create libraries that you can share among programs.
- Distribute libraries to other developers without making the source code available.

This topic covers the overall structure of a Delphi application: the **program** header, **unit** declaration syntax, and the **uses** clause. Specific differences between the Win32 and .NET platforms are noted in the text. The Delphi compiler does not support .NET namespaces on the Win32 platform. The RAD Studio compiler does support hierarchical .NET namespaces; this topic is covered in the following section, Using Namespaces with Delphi.

Program Structure and Syntax

A complete, executable Delphi application consists of multiple unit modules, all tied together by a single source code module called a project file. In traditional Pascal programming, all source code, including the main program, is stored in `.pas` files. CodeGear tools use the file extension `.dpr` to designate the main program source module, while most other source code resides in unit files having the traditional `.pas` extension. To build a project, the compiler needs the project source file, and either a source file or a compiled unit file for each unit.

Note: Strictly speaking, you need not explicitly use any units in a project, but all programs automatically use the `System` unit and the `SysInit` unit.

The source code file for an executable Delphi application contains

- a **program** heading,
- a **uses** clause (optional), and
- a block of declarations and executable statements.

Additionally, a RAD Studio program may contain a **namespaces** clause, to specify additional namespaces in which to search for generic units. This topic is covered in more detail in the section Using .NET Namespaces with Delphi.

The compiler, and hence the IDE, expect to find these three elements in a single project (.dpr) file.

The Program Heading

The **program** heading specifies a name for the executable program. It consists of the reserved word **program**, followed by a valid identifier, followed by a semicolon. For applications developed using CodeGear tools, the identifier must match the project source file name.

The following example shows the project source file for a program called Editor. Since the program is called Editor, this project file is called Editor.dpr.

```
program Editor;

  uses Forms, REAbout, // An "About" box
       REMain;         // Main form

  {$R *.res}

  begin
   Application.Title := 'Text Editor';
   Application.CreateForm(TMainForm, MainForm);
   Application.Run;
  end.
```

The first line contains the **program** heading. The **uses** clause in this example specifies a dependency on three additional units: Forms, REAbout, and REMain. The $R compiler directive links the project's resource file into the program. Finally, the block of statements between the **begin** and **end** keywords are executed when the program runs. The project file, like all Delphi source files, ends with a period (not a semicolon).

Delphi project files are usually short, since most of a program's logic resides in its unit files. A Delphi project file typically contains only enough code to launch the application's main window, and start the event processing loop. Project files are generated and maintained automatically by the IDE, and it is seldom necessary to edit them manually.

In standard Pascal, a program heading can include parameters after the program name:

```
program Calc(input, output);
```

CodeGear's Delphi ignores these parameters.

In RAD Studio, a the **program** heading introduces its own namespace, which is called the project default namespace. This is also true for the **library** and **package** headers, when these types of projects are compiled for the .NET platform.

The Program Uses Clause

The **uses** clause lists those units that are incorporated into the program. These units may in turn have **uses** clauses of their own. For more information on the **uses** clause within a unit source file, see *Unit References and the Uses Clause*, below.

The uses clause consists of the keyword **uses**, followed by a comma delimited list of units the project file directly depends on.

The Block

The block contains a simple or structured statement that is executed when the program runs. In most **program** files, the block consists of a compound statement bracketed between the reserved words **begin** and **end**, whose component statements are simply method calls to the project's Application object. Most projects have a global Application variable that holds an instance of TApplication, TWebApplication, or TServiceApplication. The block can also contain declarations of constants, types, variables, procedures, and functions; these declarations must precede the statement part of the block.

Unit Structure and Syntax

A unit consists of types (including classes), constants, variables, and routines (functions and procedures). Each unit is defined in its own source (.pas) file.

A unit file begins with a **unit** heading, which is followed by the **interface** keyword. Following the **interface** keyword, the **uses** clause specifies a list of unit dependencies. Next comes the **implementation** section, followed by the optional **initialization**, and **finalization** sections. A skeleton unit source file looks like this:

```
unit Unit1;

interface

uses // List of unit dependencies goes here...

implementation

uses // List of unit dependencies goes here...

// Implementation of class methods, procedures, and functions goes here...

initialization

// Unit initialization code goes here...

finalization

// Unit finalization code goes here...

end.
```

The unit must conclude with the reserved word **end** followed by a period.

The Unit Heading

The **unit** heading specifies the unit's name. It consists of the reserved word **unit**, followed by a valid identifier, followed by a semicolon. For applications developed using CodeGear tools, the identifier must match the unit file name. Thus, the **unit** heading

```
unit MainForm;
```

would occur in a source file called `MainForm.pas`, and the file containing the compiled unit would be `MainForm.dcu` or `MainForm.dcuil`.

Unit names must be unique within a project. Even if their unit files are in different directories, two units with the same name cannot be used in a single program.

The Interface Section

The **interface** section of a unit begins with the reserved word **interface** and continues until the beginning of the **implementation** section. The **interface** section declares constants, types, variables, procedures, and functions that are available to clients. That is, to other units or programs that wish to use elements from this unit. These entities are called *public* because code in other units can access them as if they were declared in the unit itself.

The **interface** declaration of a procedure or function includes only the routine's signature. That is, the routine's name, parameters, and return type (for functions). The block containing executable code for the procedure or function follows in the **implementation** section. Thus procedure and function declarations in the interface section work like forward declarations.

The **interface** declaration for a class must include declarations for all class members: fields, properties, procedures, and functions.

The **interface** section can include its own **uses** clause, which must appear immediately after the keyword **interface**.

The Implementation Section

The **implementation** section of a unit begins with the reserved word **implementation** and continues until the beginning of the **initialization** section or, if there is no **initialization** section, until the end of the unit. The **implementation** section defines procedures and functions that are declared in the **interface** section. Within the **implementation** section, these procedures and functions may be defined and called in any order. You can omit parameter lists from public procedure and function headings when you define them in the **implementation** section; but if you include a parameter list, it must match the declaration in the **interface** section exactly.

In addition to definitions of public procedures and functions, the **implementation** section can declare constants, types (including classes), variables, procedures, and functions that are *private* to the unit. That is, unlike the **interface** section, entities declared in the **implementation** section are inaccessible to other units.

The **implementation** section can include its own **uses** clause, which must appear immediately after the keyword **implementation**. The identifiers declared within units specified in the **implementation** section are only available for use within the **implementation** section itself. You cannot refer to such identifiers in the **interface** section.

The Initialization Section

The **initialization** section is optional. It begins with the reserved word **initialization** and continues until the beginning of the **finalization** section or, if there is no **finalization** section, until the end of the unit. The **initialization** section contains statements that are executed, in the order in which they appear, on program start-up. So, for example, if you have defined data structures that need to be initialized, you can do this in the **initialization** section.

For units in the **interfaceuses** list, the **initialization** sections of the units used by a client are executed in the order in which the units appear in the client's **uses** clause.

The Finalization Section

The **finalization** section is optional and can appear only in units that have an **initialization** section. The **finalization** section begins with the reserved word **finalization** and continues until the end of the unit. It contains statements that are executed when the main program terminates (unless the *Halt* procedure is used to terminate the program). Use the **finalization** section to free resources that are allocated in the **initialization** section.

Finalization sections are executed in the opposite order from **initialization** sections. For example, if your application initializes units A, B, and C, in that order, it will finalize them in the order C, B, and A.

Once a unit's **initialization** code starts to execute, the corresponding **finalization** section is guaranteed to execute when the application shuts down. The **finalization** section must therefore be able to handle incompletely initialized data, since, if a runtime error occurs, the **initialization** code might not execute completely.

Note: The initialization and finalization sections behave differently when code is compiled for the managed .NET environment. See the chapter on Memory Management for more information.

Unit References and the Uses Clause

A **uses** clause lists units used by the program, library, or unit in which the clause appears. A **uses** clause can occur in

- the project file for a **program**, or **library**
- the **interface** section of a unit
- the **implementation** section of a unit

Most project files contain a **uses** clause, as do the **interface** sections of most units. The **implementation** section of a unit can contain its own **uses** clause as well.

The `System` unit and the `SysInit` unit are used automatically by every application and cannot be listed explicitly in the **uses** clause. (`System` implements routines for file I/O, string handling, floating point operations, dynamic memory allocation, and so forth.) Other standard library units, such as `SysUtils`, must be explicitly included in the **uses** clause. In most cases, all necessary units are placed in the **uses** clause by the IDE, as you add and remove units from your project.

In **unit** declarations and **uses** clauses, unit names must match the file names in case. In other contexts (such as qualified identifiers), unit names are case insensitive. To avoid problems with unit references, refer to the unit source file explicitly:

```
uses MyUnit in "myunit.pas";
```

If such an explicit reference appears in the project file, other source files can refer to the unit with a simple uses clause that does not need to match case:

```
uses Myunit;
```

The Syntax of a Uses Clause

A **uses** clause consists of the reserved word **uses**, followed by one or more comma delimited unit names, followed by a semicolon. Examples:

```
uses Forms, Main;
```

```
uses
   Forms,
   Main;
```

```
uses Windows, Messages, SysUtils, Strings, Classes, Unit2, MyUnit;
```

In the **uses** clause of a **program** or **library**, any unit name may be followed by the reserved word **in** and the name of a source file, with or without a directory path, in single quotation marks; directory paths can be absolute or relative. Examples:

```
uses
   Windows, Messages, SysUtils,
   Strings in 'C:\Classes\Strings.pas', Classes;
```

Use the keyword **in** after a unit name when you need to specify the unit's source file. Since the IDE expects unit names to match the names of the source files in which they reside, there is usually no reason to do this. Using **in** is necessary only when the location of the source file is unclear, for example when

- You have used a source file that is in a different directory from the project file, and that directory is not in the compiler's search path.
- Different directories in the compiler's search path have identically named units.
- You are compiling a console application from the command line, and you have named a unit with an identifier that doesn't match the name of its source file.

The compiler also relies on the **in** ... construction to determine which units are part of a project. Only units that appear in a project (`.dpr`) file's **uses** clause followed by **in** and a file name are considered to be part of the project; other units in the **uses** clause are used by the project without belonging to it. This distinction has no effect on compilation, but it affects IDE tools like the **Project Manager**.

In the **uses** clause of a unit, you cannot use **in** to tell the compiler where to find a source file. Every unit must be in the compiler's search path. Moreover, unit names must match the names of their source files.

Multiple and Indirect Unit References

The order in which units appear in the uses clause determines the order of their initialization and affects the way identifiers are located by the compiler. If two units declare a variable, constant, type, procedure, or function with the same name, the compiler uses the one from the unit listed last in the **uses** clause. (To access the identifier from the other unit, you would have to add a qualifier: `UnitName.Identifier`.)

A **uses** clause need include only units used directly by the program or unit in which the clause appears. That is, if unit A references constants, types, variables, procedures, or functions that are declared in unit B, then A must use B explicitly. If B in turn references identifiers from unit C, then A is indirectly dependent on C; in this case, C needn't be included in a **uses** clause in A, but the compiler must still be able to find both B and C in order to process A.

The following example illustrates indirect dependency.

```
program Prog;
uses Unit2;
const a = b;
// ...
```

```
unit Unit2;
interface
uses Unit1;
const b = c;
// ...

unit Unit1;
interface
const c = 1;
// ...
```

In this example, `Prog` depends directly on `Unit2`, which depends directly on `Unit1`. Hence `Prog` is indirectly dependent on `Unit1`. Because `Unit1` does not appear in `Prog`'s **uses** clause, identifiers declared in `Unit1` are not available to `Prog`.

To compile a client module, the compiler needs to locate all units that the client depends on, directly or indirectly. Unless the source code for these units has changed, however, the compiler needs only their `.dcu` (Win32) or `.dcuil` (.NET) files, not their source (`.pas`) files.

When a change is made in the interface section of a unit, other units that depend on the change must be recompiled. But when changes are made only in the implementation or other sections of a unit, dependent units don't have to be recompiled. The compiler tracks these dependencies automatically and recompiles units only when necessary.

Circular Unit References

When units reference each other directly or indirectly, the units are said to be mutually dependent. Mutual dependencies are allowed as long as there are no circular paths connecting the **uses** clause of one interface section to the **uses** clause of another. In other words, starting from the interface section of a unit, it must never be possible to return to that unit by following references through interface sections of other units. For a pattern of mutual dependencies to be valid, each circular reference path must lead through the **uses** clause of at least one implementation section.

In the simplest case of two mutually dependent units, this means that the units cannot list each other in their interface **uses** clauses. So the following example leads to a compilation error:

```
unit Unit1;
interface
uses Unit2;
// ...

unit Unit2;
interface
uses Unit1;
// ...
```

However, the two units can legally reference each other if one of the references is moved to the implementation section:

```
unit Unit1;
interface
uses Unit2;
// ...

unit Unit2;
interface
//...

implementation
uses Unit1;
// ...
```

To reduce the chance of circular references, it's a good idea to list units in the implementation **uses** clause whenever possible. Only when identifiers from another unit are used in the interface section is it necessary to list that unit in the interface **uses** clause.

See Also

Using Namespaces with Delphi (see page 689)

3.1.3.12.2 Using Namespaces with Delphi

In Delphi, a unit is the basic container for types. Microsoft's Common Language Runtime (CLR) introduces another layer of organization called a namespace. In the .NET Framework, a namespace is a conceptual container of types. In Delphi, a namespace is a container of Delphi units. The addition of namespaces gives Delphi the ability to access and extend classes in the .NET Framework.

Unlike traditional Delphi units, namespaces can be nested to form a containment hierarchy. Nested namespaces provide a way to organize identifiers and types, and are used to disambiguate types with the same name. Since they are a container for Delphi units, namespaces may also be used to differentiate between units of the same name, that reside in different packages.

For example, the class `MyClass` in `MyNameSpace`, is different from the class `MyClass` in `YourNamespace`. At runtime, the CLR always refers to classes and types by their fully qualified names: the assembly name, followed by the namespace that contains the type. The CLR itself has no concept or implementation of the namespace hierarchy; it is purely a notational convenience of the programming language.

The following topics are covered:

- Project default namespaces, and namespace declaration.
- Namespace search scope.
- Using namespaces in Delphi units.

Declaring Namespaces

In RAD Studio, a project file (**program**, **library**, or **package**) implicitly introduces its own namespace, called the *project default namespace*. A unit may be a member of the project default namespace, or it may explicitly declare itself to be a member of a different namespace. In either case, a unit declares its namespace membership in its **unit** header. For example, consider the following explicit namespace declaration:

```
unit MyCompany.MyWidgets.MyUnit;
```

First, notice that namespaces are separated by dots. Namespaces do not introduce new symbols for the identifiers between the dots; the dots are part of the unit name. The source file name for this example is `MyCompany.MyWidgets.MyUnit.pas`, and the compiled output file name is `MyCompany.MyWidgets.MyUnit.dcuil`.

Second, notice that the dots imply the conceptual nesting, or containment, of one namespace within another. The example above declares the unit `MyUnit` to be a member of the `MyWidgets` namespace, which itself is contained in the `MyCompany` namespace. Again, it should be noted that this containment is for documentation purposes only.

A project default namespace declares a namespace for all of the units in the project. Consider the following declarations:

```
Program MyCompany.Programs.MyProgram;
Library MyCompany.Libs.MyLibrary;
Package MyCompany.Packages.MyPackage;
```

These statements establish the project default namespace for the **program**, **library**, and **package**, respectively. The namespace is determined by removing the rightmost identifier (and dot) from the declaration.

A unit that omits an explicit namespace is called a *generic unit*. A generic unit automatically becomes a member of the project default namespace. Given the preceding **program** declaration, the following unit declaration would cause the compiler to treat `MyUnit` as a member of the `MyCompany.Programs` namespace.

```
unit MyUnit;
```

The project default namespace does not affect the name of the Delphi source file for a generic unit. In the preceding example, the Delphi source file name would be `MyUnit.pas`. The compiler does however prefix the `dcuil` file name with the project default namespace. The resulting `dcuil` file in the current example would be `MyCompany.Programs.MyUnit.dcuil`.

Namespace strings are not case-sensitive. The compiler considers two namespaces that differ only in case to be equivalent. However, the compiler does preserve the case of a namespace, and will use the preserved casing in output file names, error messages, and RTTI unit identifiers. RTTI for class and type names will include the full namespace specification.

Searching Namespaces

A unit must declare the other units on which it depends. As with the Win32 platform, the compiler must search these units for identifiers. For units in explicit namespaces the search scope is already known, but for generic units, the compiler must establish a namespace search scope.

Consider the following **unit** and **uses** declarations:

```
unit MyCompany.ProjectX.ProgramY.MyUnit1;
uses MyCompany.Libs.Unit2, Unit3, Unit4;
```

These declarations establish `MyUnit1` as a member of the `MyCompany.ProjectX.ProgramY` namespace. `MyUnit1` depends on three other units: `MyCompany.Libs.Unit2`, and the generic units, `Unit3`, and `Unit4`. The compiler can resolve identifier names in `Unit2`, since the **uses** clause specified the fully qualified unit name. To resolve identifier names in `Unit3` and `Unit4`, the compiler must establish a namespace search order.

Namespace search order

Search locations can come from three possible sources: compiler options, the project default namespace, and the current unit's namespace.

The compiler resolves identifier names in the following order:

1. The current unit namespace (if any)
2. The project default namespace (if any)
3. Namespaces specified by compiler options.

A namespace search example

The following example project and unit files demonstrate the namespace search order:

```
// Project file declarations...
program MyCompany.ProjectX.ProgramY;
// Unit source file declaration...
unit MyCompany.ProjectX.ProgramY.MyUnit1;
```

Given this program example, the compiler would search namespaces in the following order:

1. `MyCompany.ProjectX.ProgramY`
2. `MyCompany.ProjectX`
3. Namespaces specified by compiler options.

Note that if the current unit is generic (i.e. it does not have an explicit namespace declaration in its **unit** statement), then resolution begins with the project default namespace.

Using Namespaces

Delphi's **uses** clause brings a module into the context of the current unit. The **uses** clause must either refer to a module by its fully qualified name (i.e. including the full namespace specification), or by its generic name, thereby relying on the namespace resolution mechanisms to locate the unit.

Fully qualified unit names

The following example demonstrates the **uses** clause with namespaces:

```
unit MyCompany.Libs.MyUnit1
uses MyCompany.Libs.Unit2,  // Fully qualified name.
     UnitX;                 // Generic name.
```

Once a module has been brought into context, source code can refer to identifiers within that module either by the unqualified name, or by the fully qualified name (if necessary, to disambiguate identifiers with the same name in different units). The following `writeln` statements are equivalent:

```
uses MyCompany.Libs.Unit2;

begin
   writeln(MyCompany.Libs.Unit2.SomeString);
   writeln(SomeString);
end.
```

A fully qualified identifier must include the full namespace specification. In the preceding example, it would be an error to refer to `SomeString` using only a portion of the namespace:

```
writeln(Unit2.SomeString);       // ERROR!
writeln(Libs.Unit2.SomeString);  // ERROR!
writeln(MyCompany.Libs.Unit2.SomeString);      // Correct.
writeln(SomeString);                           // Correct.
```

It is also an error to refer to only a portion of a namespace in the **uses** clause. There is no mechanism to import all units and symbols in a namespace. The following code does not import all units and symbols in the `MyCompany` namespace:

```
uses MyCompany;   // ERROR!
```

This restriction also applies to the **with**-**do** statement. The following will produce a compiler error:

```
with MyCompany.Libs do    // ERROR!
```

Namespaces and .NET Metadata

The Delphi for .NET compiler does not emit the entire dotted unit name into the assembly. Instead, the only leftmost portion - everything up to the last dot in the name is emitted. For example:

```
unit MyCompany.MyClasses.MyUnit
```

The compiler will emit the namespace `MyCompany.MyClasses` into the assembly metadata. This makes it easier for other .NET languages to call into Delphi assemblies.

This difference in namespace metadata is visible only to external consumers of the assembly. The Delphi code within the assembly still treats the entire dotted name as the fully qualified name.

Multi-unit Namespaces

Multiple units can belong to the same namespace, if the unit declarations refer to the same namespace. For example, one can create two files, unit1.pas and unit2.pas, with the following unit declarations:

```
// in file 'unit1.pas'
unit MyCompany.ProjectX.ProgramY.Unit1
// in file 'unit2.pas'
unit MyCompany.ProjectX.ProgramY.Unit2
```

In this example, the namespace `MyCompany.ProjectX.ProgramY` logically contains all of the **interface** symbols from `unit1.pas` and `unit2.pas`.

Symbol names in a namespace must be unique, across all units in the namespace. In the example above, it is an error for

`Unit1` and `Unit2` to both define a global interface symbol named `mySymbol`.

The individual units aggregated in a namespace are not available to source code unless the individual units are explicitly used in the file's **uses** clause. In other words, if a source file uses only the namespace, then fully qualified identifier expressions referring to a symbol in a unit in that namespace must use the namespace name, not just the name of the unit that defines that symbol.

A **uses** clause may refer to a namespace as well as individual units within that namespace. In this case, a fully qualified expression referring to a symbol from a specific unit listed in the **uses** clause may be referred to using the actual unit name or the fully-qualified name (including namespace and unit name) for the qualifier. The two forms of reference are identical and refer to the same symbol.

Note: Explicitly using a unit in the uses

clause will only work when you are compiling from source or `dcu` files. If the namespace units are compiled into an assembly and the assembly is referenced by the project instead of the individual units, then the source code that explicitly refers to a unit in the namespace will fail.

See Also

Programs and Units (see page 683)

3.1.3.13 Standard Routines and I/O

This section describes the standard routines included in the Delphi runtime library.

Topics

Name	Description
Standard Routines and I/O (see page 692)	These topics discuss text and file I/O and summarize standard library routines. Many of the procedures and functions listed here are defined in the `System` and *SysInit* units, which are implicitly used with every application. Others are built into the compiler but are treated as if they were in the `System` unit. Some standard routines are in units such as `SysUtils`, which must be listed in a **uses** clause to make them available in programs. You cannot, however, list `System` in a **uses** clause, nor should you modify the `System` unit or try to rebuild it explicitly.

3.1.3.13.1 Standard Routines and I/O

These topics discuss text and file I/O and summarize standard library routines. Many of the procedures and functions listed here are defined in the `System` and *SysInit* units, which are implicitly used with every application. Others are built into the compiler but are treated as if they were in the `System` unit.

Some standard routines are in units such as `SysUtils`, which must be listed in a **uses** clause to make them available in programs. You cannot, however, list `System` in a **uses** clause, nor should you modify the `System` unit or try to rebuild it explicitly.

File Input and Output

The table below lists input and output routines.

Input and output procedures and functions

Procedure or function	Description
Append	Opens an existing text file for appending.
AssignFile	Assigns the name of an external file to a file variable.
BlockRead	Reads one or more records from an untyped file.
BlockWrite	Writes one or more records into an untyped file.

ChDir	Changes the current directory.
CloseFile	Closes an open file.
Eof	Returns the end-of-file status of a file.
Eoln	Returns the end-of-line status of a text file.
Erase	Erases an external file.
FilePos	Returns the current file position of a typed or untyped file.
FileSize	Returns the current size of a file; not used for text files.
Flush	Flushes the buffer of an output text file.
GetDir	Returns the current directory of a specified drive.
IOResult	Returns an integer value that is the status of the last I/O function performed.
MkDir	Creates a subdirectory.
Read	Reads one or more values from a file into one or more variables.
Readln	Does what Read does and then skips to beginning of next line in the text file.
Rename	Renames an external file.
Reset	Opens an existing file.
Rewrite	Creates and opens a new file.
RmDir	Removes an empty subdirectory.
Seek	Moves the current position of a typed or untyped file to a specified component. Not used with text files.
SeekEof	Returns the end-of-file status of a text file.
SeekEoln	Returns the end-of-line status of a text file.
SetTextBuf	Assigns an I/O buffer to a text file.
Truncate	Truncates a typed or untyped file at the current file position.
Write	Writes one or more values to a file.
Writeln	Does the same as Write, and then writes an end-of-line marker to the text file.

A file variable is any variable whose type is a file type. There are three classes of file: typed, text, and untyped. The syntax for declaring file types is given in File types. Note that file types are only available on the Win32 platform.

Before a file variable can be used, it must be associated with an external file through a call to the AssignFile procedure. An external file is typically a named disk file, but it can also be a device, such as the keyboard or the display. The external file stores the information written to the file or supplies the information read from the file.

Once the association with an external file is established, the file variable must be *opened* to prepare it for input or output. An existing file can be opened via the Reset procedure, and a new file can be created and opened via the Rewrite procedure. Text files opened with Reset are read-only and text files opened with Rewrite and Append are write-only. Typed files and untyped files always allow both reading and writing regardless of whether they were opened with Reset or Rewrite.

Every file is a linear sequence of components, each of which has the component type (or record type) of the file. The components are numbered starting with zero.

Files are normally accessed sequentially. That is, when a component is read using the standard procedure Read or written using the standard procedure Write, the current file position moves to the next numerically ordered file component. Typed files and untyped files can also be accessed randomly through the standard procedure Seek, which moves the current file position to a specified component. The standard functions FilePos and FileSize can be used to determine the current file position and the current file size.

3

When a program completes processing a file, the file must be closed using the standard procedure CloseFile. After a file is closed, its associated external file is updated. The file variable can then be associated with another external file.

By default, all calls to standard I/O procedures and functions are automatically checked for errors, and if an error occurs an exception is raised (or the program is terminated if exception handling is not enabled). This automatic checking can be turned on and off using the **{$I+}** and **{$I-}** compiler directives. When I/O checking is off, that is, when a procedure or function call is compiled in the **{$I-}** state an I/O error doesn't cause an exception to be raised; to check the result of an I/O operation, you must call the standard function IOResult instead.

You must call the IOResult function to clear an error, even if you aren't interested in the error. If you don't clear an error and **{$I-}** is the current state, the next I/O function call will fail with the lingering IOResult error.

Text Files

This section summarizes I/O using file variables of the standard type Text.

When a text file is opened, the external file is interpreted in a special way: It is considered to represent a sequence of characters formatted into lines, where each line is terminated by an end-of-line marker (a carriage-return character, possibly followed by a line feed character). The type Text is distinct from the type file of Char.

For text files, there are special forms of Read and Write that let you read and write values that are not of type Char. Such values are automatically translated to and from their character representation. For example, `Read(F, I)`, where I is a type Integer variable, reads a sequence of digits, interprets that sequence as a decimal integer, and stores it in I.

There are two standard text file variables, Input and Output The standard file variable Input is a read-only file associated with the operating system's standard input (typically, the keyboard). The standard file variable Output is a write-only file associated with the operating system's standard output (typically, the display). Before an application begins executing, Input and Output are automatically opened, as if the following statements were executed:

```
AssignFile(Input, '');
Reset(Input);
AssignFile(Output, '');
Rewrite(Output);
```

Note: For Win32 applications, text-oriented I/O is available only in console applications, that is, applications compiled with the Generate console application

option checked on the Linker page of the Project Options dialog box or with the **-cc** command-line compiler option. In a GUI (non-console) application, any attempt to read or write using Input or Output will produce an I/O error.

Some of the standard I/O routines that work on text files don't need to have a file variable explicitly given as a parameter. If the file parameter is omitted, Input or Output is assumed by default, depending on whether the procedure or function is input- or output-oriented. For example, `Read(X)` corresponds to `Read(Input, X)` and `Write(X)` corresponds to `Write(Output, X)`.

If you do specify a file when calling one of the input or output routines that work on text files, the file must be associated with an external file using AssignFile, and opened using Reset, Rewrite, or Append. An error occurs if you pass a file that was opened with Reset to an output-oriented procedure or function. An error also occurs if you pass a file that was opened with Rewrite or Append to an input-oriented procedure or function.

Untyped Files

Untyped files are low-level I/O channels used primarily for direct access to disk files regardless of type and structuring. An untyped file is declared with the word **file** and nothing more. For example,

```
var DataFile: file;
```

For untyped files, the Reset and Rewrite procedures allow an extra parameter to specify the record size used in data transfers. For historical reasons, the default record size is 128 bytes. A record size of 1 is the only value that correctly reflects the exact size of any file. (No partial records are possible when the record size is 1.)

Except for Read and Write, all typed-file standard procedures and functions are also allowed on untyped files. Instead of Read and Write, two procedures called BlockRead and BlockWrite are used for high-speed data transfers.

Text File Device Drivers

You can define your own text file device drivers for your programs. A text file device driver is a set of four functions that completely implement an interface between Delphi's file system and some device.

The four functions that define each device driver are `Open`, `InOut`, `Flush`, and `Close`. The function header of each function is

```
function DeviceFunc(var F: TTextRec): Integer;
```

where `DeviceFunc` is the name of the function (that is, `Open`, `InOut`, `Flush`, or `Close`). The return value of a device-interface function becomes the value returned by IOResult. If the return value is zero, the operation was successful.

To associate the device-interface functions with a specific file, you must write a customized `Assign` procedure. The `Assign` procedure must assign the addresses of the four device-interface functions to the four function pointers in the text file variable. In addition, it should store the fmClosed*magic* constant in the Mode field, store the size of the text file buffer in `BufSize`, store a pointer to the text file buffer in `BufPtr`, and clear the `Name` string.

Assuming, for example, that the four device-interface functions are called `DevOpen`, `DevInOut`, `DevFlush`, and `DevClose`, the `Assign` procedure might look like this:

```
procedure AssignDev(var F: Text);
  begin
   with TTextRec(F) do
   begin
    Mode := fmClosed;
    BufSize := SizeOf(Buffer);
    BufPtr := @Buffer;
    OpenFunc := @DevOpen;
    InOutFunc := @DevInOut;
    FlushFunc := @DevFlush;
    CloseFunc := @DevClose;
    Name[0] := #0;
   end;
  end;
```

The device-interface functions can use the UserData field in the file record to store private information. This field isn't modified by the product file system at any time.

The Open function

The Open function is called by the Reset, Rewrite, and Append standard procedures to open a text file associated with a device. On entry, the `Mode` field contains fmInput, fmOutput, or fmInOut to indicate whether the `Open` function was called from Reset, Rewrite, or Append.

The Open function prepares the file for input or output, according to the `Mode` value. If `Mode` specified fmInOut (indicating that `Open` was called from Append), it must be changed to fmOutput before `Open` returns.

`Open` is always called before any of the other device-interface functions. For that reason, `AssignDev` only initializes the `OpenFunc` field, leaving initialization of the remaining vectors up to `Open`. Based on `Mode`, `Open` can then install pointers to either input- or output-oriented functions. This saves the `InOut`, Flush functions and the CloseFile procedure from determining the current mode.

The InOut function

The `InOut` function is called by the Read, Readln, Write, Writeln, Eof, Eoln, SeekEof, SeekEoln, and CloseFile standard routines whenever input or output from the device is required.

When `Mode` is fmInput, the `InOut` function reads up to `BufSize` characters into `BufPtr^`, and returns the number of characters read in `BufEnd`. In addition, it stores zero in `BufPos`. If the `InOut` function returns zero in `BufEnd` as a result of an input request, `Eof` becomes **True** for the file.

When `Mode` is fmOutput, the `InOut` function writes `BufPos` characters from `BufPtr^`, and returns zero in `BufPos`.

The Flush function

The `Flush` function is called at the end of each Read, Readln, Write, and Writeln. It can optionally flush the text file buffer.

If `Mode` is fmInput, the `Flush` function can store zero in `BufPos` and `BufEnd` to flush the remaining (unread) characters in the buffer. This feature is seldom used.

If `Mode` is fmOutput, the `Flush` function can write the contents of the buffer exactly like the `InOut` function, which ensures that text written to the device appears on the device immediately. If `Flush` does nothing, the text doesn't appear on the device until the buffer becomes full or the file is closed.

The Close function

The `Close` function is called by the CloseFile standard procedure to close a text file associated with a device. (The Reset, Rewrite, and Append procedures also call Close if the file they are opening is already open.) If `Mode` is fmOutput, then before calling Close, the file system calls the `InOut` function to ensure that all characters have been written to the device.

Handling null-Terminated Strings

The Delphi language's extended syntax allows the Read, Readln, Str, and Val standard procedures to be applied to zero-based character arrays, and allows the Write, Writeln, Val, AssignFile, and Rename standard procedures to be applied to both zero-based character arrays and character pointers.

Null-Terminated String Functions

The following functions are provided for handling null-terminated strings.

Null-terminated string functions

Function	Description
StrAlloc	Allocates a character buffer of a given size on the heap.
StrBufSize	Returns the size of a character buffer allocated using StrAlloc or StrNew.
StrCat	Concatenates two strings.
StrComp	Compares two strings.
StrCopy	Copies a string.
StrDispose	Disposes a character buffer allocated using StrAlloc or StrNew.
StrECopy	Copies a string and returns a pointer to the end of the string.
StrEnd	Returns a pointer to the end of a string.
StrFmt	Formats one or more values into a string.
StrIComp	Compares two strings without case sensitivity.
StrLCat	Concatenates two strings with a given maximum length of the resulting string.
StrLComp	Compares two strings for a given maximum length.
StrLCopy	Copies a string up to a given maximum length.
StrLen	Returns the length of a string.
StrLFmt	Formats one or more values into a string with a given maximum length.
StrLIComp	Compares two strings for a given maximum length without case sensitivity.
StrLower	Converts a string to lowercase.

StrMove	Moves a block of characters from one string to another.
StrNew	Allocates a string on the heap.
StrPCopy	Copies a Pascal string to a null-terminated string.
StrPLCopy	Copies a Pascal string to a null-terminated string with a given maximum length.
StrPos	Returns a pointer to the first occurrence of a given substring within a string.
StrRScan	Returns a pointer to the last occurrence of a given character within a string.
StrScan	Returns a pointer to the first occurrence of a given character within a string.
StrUpper	Converts a string to uppercase.

Standard string-handling functions have multibyte-enabled counterparts that also implement locale-specific ordering for characters. Names of multibyte functions start with Ansi-. For example, the multibyte version of StrPos is AnsiStrPos. Multibyte character support is operating-system dependent and based on the current locale.

Wide-Character Strings

The `System` unit provides three functions, WideCharToString, WideCharLenToString, and StringToWideChar, that can be used to convert null-terminated wide character strings to single- or double-byte long strings.

Assignment will also convert between strings. For instance, the following are both valid:

```
MyAnsiString := MyWideString;
     MyWideString := MyAnsiString;
```

Other Standard Routines

The table below lists frequently used procedures and functions found in CodeGear product libraries. This is not an exhaustive inventory of standard routines.

Other standard routines

Procedure or function	Description
Addr	Returns a pointer to a specified object.
AllocMem	Allocates a memory block and initializes each byte to zero.
ArcTan	Calculates the arctangent of the given number.
Assert	Raises an exception if the passed expression does not evaluate to true.
Assigned	Tests for a **nil** (unassigned) pointer or procedural variable.
Beep	Generates a standard beep.
Break	Causes control to exit a **for**, **while**, or **repeat** statement.
ByteToCharIndex	Returns the position of the character containing a specified byte in a string.
Chr	Returns the character for a specified integer value.
Close	Closes a file.
CompareMem	Performs a binary comparison of two memory images.
CompareStr	Compares strings case sensitively.
CompareText	Compares strings by ordinal value and is not case sensitive.
Continue	Returns control to the next iteration of **for**, **while**, or **repeat** statements.
Copy	Returns a substring of a string or a segment of a dynamic array.

Cos	Calculates the cosine of an angle.
CurrToStr	Converts a currency variable to a string.
Date	Returns the current date.
DateTimeToStr	Converts a variable of type TDateTime to a string.
DateToStr	Converts a variable of type TDateTime to a string.
Dec	Decrements an ordinal variable or a typed pointer variable.
Dispose	Releases dynamically allocated variable memory.
ExceptAddr	Returns the address at which the current exception was raised.
Exit	Exits from the current procedure.
Exp	Calculates the exponential of X.
FillChar	Fills contiguous bytes with a specified value.
Finalize	ninitializes a dynamically allocated variable.
FloatToStr	Converts a floating point value to a string.
FloatToStrF	Converts a floating point value to a string, using specified format.
FmtLoadStr	Returns formatted output using a resourced format string.
FmtStr	Assembles a formatted string from a series of arrays.
Format	Assembles a string from a format string and a series of arrays.
FormatDateTime	Formats a date-and-time value.
FormatFloat	Formats a floating point value.
FreeMem	Releases allocated memory.
GetMem	Allocates dynamic memory and a pointer to the address of the block.
Halt	Initiates abnormal termination of a program.
Hi	Returns the high-order byte of an expression as an unsigned value.
High	Returns the highest value in the range of a type, array, or string.
Inc	Increments an ordinal variable or a typed pointer variable.
Initialize	Initializes a dynamically allocated variable.
Insert	Inserts a substring at a specified point in a string.
Int	Returns the integer part of a real number.
IntToStr	Converts an integer to a string.
Length	Returns the length of a string or array.
Lo	Returns the low-order byte of an expression as an unsigned value.
Low	Returns the lowest value in the range of a type, array, or string.
LowerCase	Converts an ASCII string to lowercase.
MaxIntValue	Returns the largest signed value in an integer array.
MaxValue	Returns the largest signed value in an array.
MinIntValue	Returns the smallest signed value in an integer array.
MinValue	Returns smallest signed value in an array.
New	Creates a dynamic allocated variable memory and references it with a specified pointer.
Now	Returns the current date and time.

Ord	Returns the ordinal integer value of an ordinal-type expression.
Pos	Returns the index of the first single-byte character of a specified substring in a string.
Pred	Returns the predecessor of an ordinal value.
Ptr	Converts a value to a pointer.
Random	Generates random numbers within a specified range.
ReallocMem	Reallocates a dynamically allocatable memory.
Round	Returns the value of a real rounded to the nearest whole number.
SetLength	Sets the dynamic length of a string variable or array.
SetString	Sets the contents and length of the given string.
ShowException	Displays an exception message with its address.
ShowMessage	Displays a message box with an unformatted string and an OK button.
ShowMessageFmt	Displays a message box with a formatted string and an OK button.
Sin	Returns the sine of an angle in radians.
SizeOf	Returns the number of bytes occupied by a variable or type.
Sqr	Returns the square of a number.
Sqrt	Returns the square root of a number.
Str	Converts an integer or real number into a string.
StrToCurr	Converts a string to a currency value.
StrToDate	Converts a string to a date format (TDateTime).
StrToDateTime	Converts a string to a TDateTime.
StrToFloat	Converts a string to a floating-point value.
StrToInt	Converts a string to an integer.
StrToTime	Converts a string to a time format (TDateTime).
StrUpper	Returns an ASCII string in upper case.
Succ	Returns the successor of an ordinal value.
Sum	Returns the sum of the elements from an array.
Time	Returns the current time.
TimeToStr	Converts a variable of type TDateTime to a string.
Trunc	Truncates a real number to an integer.
UniqueString	Ensures that a string has only one reference. (The string may be copied to produce a single reference.)
UpCase	Converts a character to uppercase.
UpperCase	Returns a string in uppercase.
VarArrayCreate	Creates a variant array.
VarArrayDimCount	Returns number of dimensions of a variant array.
VarArrayHighBound	Returns high bound for a dimension in a variant array.
VarArrayLock	Locks a variant array and returns a pointer to the data.
VarArrayLowBound	Returns the low bound of a dimension in a variant array.
VarArrayOf	Creates and fills a one-dimensional variant array.
VarArrayRedim	Resizes a variant array.

VarArrayRef	Returns a reference to the passed variant array.
VarArrayUnlock	Unlocks a variant array.
VarAsType	Converts a variant to specified type.
VarCast	Converts a variant to a specified type, storing the result in a variable.
VarClear	Clears a variant.
VarCopy	Copies a variant.
VarToStr	Converts variant to string.
VarType	Returns type code of specified variant.

See Also

Data Types (see page 553)

Porting VCL Applications to Delphi for .NET

3.1.3.14 Fundamental Syntactic Elements

This section describes the fundamental syntactic elements, or the building blocks of the Delphi language.

Topics

Name	Description
Fundamental Syntactic Elements (see page 701)	This topic introduces the Delphi language character set, and describes the syntax for declaring: • Identifiers • Numbers • Character strings • Labels • Source code comments
Declarations and Statements (see page 705)	This topic describes the syntax of Delphi declarations and statements. Aside from the **uses** clause (and reserved words like **implementation** that demarcate parts of a unit), a program consists entirely of *declarations* and *statements*, which are organized into *blocks*. This topic covers the following items: • Declarations • Simple statements such as assignment • Structured statements such as conditional tests (e.g., `if-then`, and **case**), iteration (e.g., `for`, and `while`). string; The syntax and placement of a declaration depend on the kind of identifier you are defining. In general, declarations can occur only at the beginning of a... more (see page 705)
Expressions (see page 720)	circuit evaluation is usually preferable because it guarantees minimum execution time and, in most cases, minimum code size. Complete evaluation is sometimes convenient when one operand is a function with side effects that alter the execution of the program. circuit evaluation also allows the use of constructions that might otherwise result in illegal runtime operations. For example, the following code iterates through the string S, up to the first comma.

3.1.3.14.1 Fundamental Syntactic Elements

This topic introduces the Delphi language character set, and describes the syntax for declaring:

- Identifiers
- Numbers
- Character strings
- Labels
- Source code comments

The Delphi Character Set

The Delphi Language uses the Unicode character set, including alphabetic and alphanumeric Unicode characters and the underscore. It is not case-sensitive. The space character and the ASCII control characters (ASCII 0 through 31 including ASCII 13, the return or end-of-line character) are called *blanks*.

The RAD Studio compiler will accept a file encoded in UCS-2 or UCS-4 if the file contains a byte order mark. The speed of compilation may be penalized by the use for formats other than UTF–8, however. All characters in a UCS-4 encoded source file must be representable in UCS-2 without surrogate pairs. UCS-2 encodings with surrogate pairs (including GB18030) are accepted only if the `codepage` compiler option is specified.

Fundamental syntactic elements, called *tokens*, combine to form expressions, declarations, and statements. A *statement* describes an algorithmic action that can be executed within a program. An *expression* is a syntactic unit that occurs within a statement and denotes a value. A *declaration* defines an identifier (such as the name of a function or variable) that can be used in expressions and statements, and, where appropriate, allocates memory for the identifier.

The Delphi Character Set and Basic Syntax

On the simplest level, a program is a sequence of tokens delimited by separators. A token is the smallest meaningful unit of text in a program. A separator is either a blank or a comment. Strictly speaking, it is not always necessary to place a separator between two tokens; for example, the code fragment

```
Size:=20;Price:=10;
```

is perfectly legal. Convention and readability, however, dictate that we write this as

```
Size := 20;
Price := 10;
```

Tokens are categorized as special symbols, identifiers, reserved words, directives, numerals, labels, and character strings. A separator can be part of a token only if the token is a character string. Adjacent identifiers, reserved words, numerals, and labels must have one or more separators between them.

Special Symbols

Special symbols are non-alphanumeric characters, or pairs of such characters, that have fixed meanings. The following single characters are special symbols:

```
# $ & ' ( ) * + , - . / : ; < = > @ [ ] ^ { }
```

The following character pairs are also special symbols:

```
(* (. *) .) .. // := <= >= <>
```

The following table shows equivalent symbols:

Special symbol	Equivalent symbols
[	(.

]	.)
{	(*
}	*)

The left bracket **[** is equivalent to the character pair of left parenthesis and period **(.**

The right bracket **]** is equivalent to the character pair of period and right parenthesis **.)**

The left brace **{** is equivalent to the character pair of left parenthesis and asterisk **(***.

The right brace **}** is equivalent to the character pair of right parenthesis and asterisk ***)**

Note: **%**, **?**, ****, **!**, **"** (double quotation marks), _ (underscore), **|** (pipe), and **~** (tilde) are not special characters.

Identifiers

Identifiers denote constants, variables, fields, types, properties, procedures, functions, programs, units, libraries, and packages. An identifier can be of any length, but only the first 255 characters are significant. An identifier must begin with an alphabetic character or an underscore (_) and cannot contain spaces; alphanumeric characters, digits, and underscores are allowed after the first character. Reserved words cannot be used as identifiers.

Note: The .NET SDK recommends against using leading underscores in identifiers, as this pattern is reserved for system use.

Since the Delphi Language is case-insensitive, an identifier like `CalculateValue` could be written in any of these ways:

```
CalculateValue
calculateValue
calculatevalue
CALCULATEVALUE
```

Since unit names correspond to file names, inconsistencies in case can sometimes affect compilation. For more information, see the topic, *Unit References and the Uses Clause*.

Qualified Identifiers

When you use an identifier that has been declared in more than one place, it is sometimes necessary to qualify the identifier. The syntax for a qualified identifier is

identifier1.identifier2

where *identifier1* qualifies *identifier2*. For example, if two units each declare a variable called `CurrentValue`, you can specify that you want to access the `CurrentValue` in `Unit2` by writing

```
Unit2.CurrentValue
```

Qualifiers can be iterated. For example,

```
Form1.Button1.Click
```

calls the `Click` method in `Button1` of `Form1`.

If you don't qualify an identifier, its interpretation is determined by the rules of scope described in Blocks and scope (↗ see page 705).

Extended Identifiers

Particularly when programming with Delphi for .NET, you might encounter identifiers (e.g. types, or methods in a class) having the same name as a Delphi language keyword. For example, a class might have a method called `begin`. Another example is the CLR class called Type, in the System namespace. **Type** is a Delphi language keyword, and cannot be used for an identifier name.

If you qualify the identifier with its full namespace specification, then there is no problem. For example, to use the Type class, you must use its fully qualified name:

```
var
            TMyType : System.Type; // Using fully qualified namespace
                                   // avoides ambiguity with Delphi language keyword.
```

As a shorter alternative, the ampersand (**&**) operator can be used to resolve ambiguities between identifiers and Delphi language keywords. If you encounter a method or type that is the same name as a Delphi keyword, you can omit the namespace specification if you prefix the identifier name with an ampersand. For example, the following code uses the ampersand to disambiguate the CLR Type class from the Delphi keyword **type**

```
var
           TMyType : &Type; // Prefix with '&' is ok.
```

Reserved Words

The following reserved words cannot be redefined or used as identifiers.

Reserved Words

add	**else**	**initialization**	**program**	**then**
and	**end**	inline	**property**	**threadvar**
array	**except**	**interface**	**raise**	**to**
as	**exports**	**is**	**record**	**try**
asm	**file**	**label**	**remove**	**type**
begin	**final**	**library**	repeat	**unit**
case	**finalization**	**mod**	**resourcestring**	**unsafe**
class	**finally**	**nil**	**seled**	**until**
const	**for**	**not**	**set**	**uses**
constructor	**function**	**not**	**shl**	**var**
destructor	**goto**	**of**	**shr**	**while**
dispinterface	**if**	**or**	**static**	**with**
div	**implementation**	**out**	**strict private**	**xor**
do	**in**	**packed**	**strict protected**	
downto	**inherited**	**procedure**	**string**	

In addition to the words above, **private**, **protected**, **public**, **published**, and **automated** act as reserved words within class type declarations, but are otherwise treated as directives. The words **at** and **on** also have special meanings, and should be treated as reserved words.

Directives

Directives are words that are sensitive in specific locations within source code. Directives have special meanings in the Delphi language, but, unlike reserved words, appear only in contexts where user-defined identifiers cannot occur. Hence -- although it is inadvisable to do so -- you can define an identifier that looks exactly like a directive.

Directives

absolute	**dynamic**	**local**	**platform**	**requires**
abstract	**export**	**message**	**private**	**resident**

assembler	**external**	**name**	**protected**	**safecall**
automated	**far**	**near**	**public**	**stdcall**
cdecl	**forward**	**nodefault**	**published**	**stored**
contains	**implements**	**overload**	**read**	**varargs**
default	**index**	**override**	**readonly**	**virtual**
deprecated	**inline**	**package**	**register**	**write**
dispid	**library**	**pascal**	**reintroduce**	**writeonly**

Numerals

Integer and real constants can be represented in decimal notation as sequences of digits without commas or spaces, and prefixed with the **+** or **-** operator to indicate sign. Values default to positive (so that, for example, 67258 is equivalent to +67258) and must be within the range of the largest predefined real or integer type.

Numerals with decimal points or exponents denote reals, while other numerals denote integers. When the character E or e occurs within a real, it means "times ten to the power of". For example, 7E2 means 7 * 10^2, and 12.25e+6 and 12.25e6 both mean 12.25 * 10^6.

The dollar-sign prefix indicates a hexadecimal numeral, for example, $8F. Hexadecimal numbers without a preceding **-** unary operator are taken to be positive values. During an assignment, if a hexadecimal value lies outside the range of the receiving type an error is raised, except in the case of the **Integer** (32-bit integer) where a warning is raised. In this case, values exceeding the positive range for **Integer** are taken to be negative numbers in a manner consistent with 2's complement integer representation.

For more information about real and integer types, see Data Types (see page 553). For information about the data types of numerals, see True constants (see page 589).

Labels

A label is a standard Delphi language identifier with the exception that, unlike other identifiers, labels can start with a digit. Numeric labels can include no more than ten digits - that is, a numeral between 0 and 9999999999.

Labels are used in **goto** statements. For more information about **goto** statements and labels, see Goto statements (see page 705).

Character Strings

A character string, also called a string literal or string constant, consists of a quoted string, a control string, or a combination of quoted and control strings. Separators can occur only within quoted strings.

A quoted string is a sequence of up to 255 characters from the extended ASCII character set, written on one line and enclosed by apostrophes. A quoted string with nothing between the apostrophes is a null string. Two sequential apostrophes in a quoted string denote a single character, namely an apostrophe. For example,

```
'CodeGear' { CodeGear }
'You''ll see' { You'll see }
''''          { ' }
''            { null string }
' '           { a space }
```

A control string is a sequence of one or more control characters, each of which consists of the **#** symbol followed by an unsigned integer constant from 0 to 255 (decimal or hexadecimal) and denotes the corresponding ASCII character. The control string

```
#89#111#117
```

is equivalent to the quoted string

```
'You'
```

You can combine quoted strings with control strings to form larger character strings. For example, you could use

```
'Line 1'#13#10'Line 2'
```

to put a carriage-returnline-feed between 'Line 1' and 'Line 2'. However, you cannot concatenate two quoted strings in this way, since a pair of sequential apostrophes is interpreted as a single character. (To concatenate quoted strings, use the **+** operator or simply combine them into a single quoted string.)

A character string's length is the number of characters in the string. A character string of any length is compatible with any string type and with the **PChar** type. A character string of length 1 is compatible with any character type, and, when extended syntax is enabled (with compiler directive `{$X+}`), a nonempty character string of length n is compatible with zero-based arrays and packed arrays of n characters. For more information, see Datatypes (see page 553).

Comments and Compiler Directives

Comments are ignored by the compiler, except when they function as separators (delimiting adjacent tokens) or compiler directives.

There are several ways to construct comments:

```
{ Text between a left brace and a right brace constitutes a comment. }
          (* Text between a left-parenthesis-plus-asterisk and an
asterisk-plus-right-parenthesis is also a comment *)
          // Any text between a double-slash and the end of the line constitutes a comment.
```

Comments that are alike cannot be nested. For instance, `{{}}` will not work, but `(*{}*)`will. This is useful for commenting out sections of code that also contain comments.

A comment that contains a dollar sign (**$**) immediately after the opening **{** or **(*** is a compiler directive. For example,

```
{$WARNINGS OFF}
```

tells the compiler not to generate warning messages.

See Also

Expressions (see page 720)

Declarations and Statements (see page 705)

Unit References and the Uses Clause (see page 683)

3.1.3.14.2 Declarations and Statements

This topic describes the syntax of Delphi declarations and statements.

Aside from the **uses** clause (and reserved words like **implementation** that demarcate parts of a unit), a program consists entirely of *declarations* and *statements*, which are organized into *blocks*.

This topic covers the following items:

- Declarations
- Simple statements such as assignment
- Structured statements such as conditional tests (e.g., `if-then`, and **case**), iteration (e.g., `for`, and `while`).

string;

The syntax and placement of a declaration depend on the kind of identifier you are defining. In general, declarations can occur only at the beginning of a block or at the beginning of the interface or **implementation** section of a unit (after the **uses** clause). Specific conventions for declaring variables, constants, types, functions, and so forth are explained in the documentation for those topics.

Hinting Directives

The 'hint' directives **platform**, **deprecated**, and **library** may be appended to any declaration. These directives will produce warnings at compile time. Hint directives can be applied to type declarations, variable declarations, class, interface and structure declarations, field declarations within classes or records, procedure, function and method declarations, and unit declarations.

When a hint directive appears in a unit declaration, it means that the hint applies to everything in the unit. For example, the Windows 3.1 style `OleAuto.pas` unit on Windows is completely deprecated. Any reference to that unit or any symbol in that unit will produce a deprecation message.

The **platform** hinting directive on a symbol or unit indicates that it may not exist or that the implementation may vary considerably on different platforms. The **library** hinting directive on a symbol or unit indicates that the code may not exist or the implementation may vary considerably on different library architectures.

The **platform** and **library** directives do not specify which platform or library. If your goal is writing platform-independent code, you do not need to know which platform a symbol is specific to; it is sufficient that the symbol be marked as specific to *some* platform to let you know it may cause problems for your goal of portability.

In the case of a procedure or function declaration, the hint directive should be separated from the rest of the declaration with a semicolon. Examples:

```
procedure SomeOldRoutine; stdcall deprecated;
```

```
var
   VersionNumber: Real library;
```

```
type
   AppError = class(Exception)
     ...
end platform;
```

When source code is compiled in the `{$HINTS ON} {$WARNINGS ON}` state, each reference to an identifier declared with one of these directives generates an appropriate hint or warning. Use **platform** to mark items that are specific to a particular operating environment (such as Windows or .NET), **deprecated** to indicate that an item is obsolete or supported only for backward compatibility, and **library** to flag dependencies on a particular library or component framework.

The RAD Studio compiler also recognizes the hinting directive **experimental**. You can use this directive to designate units which are in an unstable, development state. The compiler will emit a warning when it builds an application that uses the unit.

Declarations

The names of variables, constants, types, fields, properties, procedures, functions, programs, units, libraries, and packages are called *identifiers*. (Numeric constants like 26057 are not identifiers.) Identifiers must be declared before you can use them; the only exceptions are a few predefined types, routines, and constants that the compiler understands automatically, the variable `Result` when it occurs inside a function block, and the variable `Self` when it occurs inside a method implementation.

A declaration defines an identifier and, where appropriate, allocates memory for it. For example,

```
var Size: Extended;
```

declares a variable called `Size` that holds an **Extended** (real) value, while

```
function DoThis(X, Y: string): Integer;
```

declares a function called `DoThis` that takes two strings as arguments and returns an integer. Each declaration ends with a semicolon. When you declare several variables, constants, types, or labels at the same time, you need only write the appropriate reserved word once:

```
var
   Size: Extended;
   Quantity: Integer;
```

Statements

Statements define algorithmic actions within a program. Simple statements like assignments and procedure calls can combine to form loops, conditional statements, and other structured statements.

Multiple statements within a block, and in the initialization or finalization section of a unit, are separated by semicolons.

Simple Statements

A simple statement doesn't contain any other statements. Simple statements include assignments, calls to procedures and functions, and goto jumps.

Assignment Statements

An assignment statement has the form

variable := expression

where *variable* is any variable reference, including a variable, variable typecast, dereferenced pointer, or component of a structured variable. The *expression* is any assignment-compatible expression (within a function block, variable can be replaced with the name of the function being defined. See Procedures and functions (see page 662)). The **:=** symbol is sometimes called the assignment operator.

An assignment statement replaces the current value of variable with the value of expression. For example,

```
I := 3;
```

assigns the value 3 to the variable `I`. The variable reference on the left side of the assignment can appear in the expression on the right. For example,

```
I := I + 1;
```

increments the value of `I`. Other assignment statements include

```
X := Y + Z;
Done := (I >= 1) and (I < 100);
Hue1 := [Blue, Succ(C)];
I := Sqr(J) - I  * K;
Shortint(MyChar) := 122;
TByteRec(W).Hi := 0;
MyString[I] := 'A';
SomeArray[I + 1] := P^;
TMyObject.SomeProperty := True;
```

Procedure and Function Calls

A procedure call consists of the name of a procedure (with or without qualifiers), followed by a parameter list (if required). Examples include

```
PrintHeading;
Transpose(A, N, M);
Find(Smith, William);
Writeln('Hello world!');
DoSomething();
Unit1.SomeProcedure;
TMyObject.SomeMethod(X,Y);
```

With extended syntax enabled (`{$X+}`), function calls, like calls to procedures, can be treated as statements in their own right:

```
MyFunction(X);
```

When you use a function call in this way, its return value is discarded.

For more information about procedures and functions, see Procedures and functions (see page 662).

Goto Statements

A **goto** statement, which has the form

```
goto label
```

transfers program execution to the statement marked by the specified label. To mark a statement, you must first declare the label. Then precede the statement you want to mark with the label and a colon:

label: statement

Declare labels like this:

`label`*label;*

You can declare several labels at once:

`label`*label1*, ..., *labeln;*

A label can be any valid identifier or any numeral between 0 and 9999.

The label declaration, marked statement, and **goto** statement must belong to the same block. (See Blocks and Scope, below.) Hence it is not possible to jump into or out of a procedure or function. Do not mark more than one statement in a block with the same label.

For example,

```
label StartHere;
    ...
StartHere: Beep;
goto StartHere;
```

creates an infinite loop that calls the `Beep` procedure repeatedly.

Additionally, it is not possible to jump into or out of a **try**-**finally** or **try**-**except** statement.

The **goto** statement is generally discouraged in structured programming. It is, however, sometimes used as a way of exiting from nested loops, as in the following example.

```
procedure FindFirstAnswer;
  var X, Y, Z, Count: Integer;
label FoundAnAnswer;
begin
  Count := SomeConstant;
  for X := 1 to Count do
    for Y := 1 to Count do
      for Z := 1 to Count do
        if ... { some condition holds on X, Y, and Z } then
          goto FoundAnAnswer;

        ... { Code to execute if no answer is found }
        Exit;

FoundAnAnswer:
        ... { Code to execute when an answer is found }
end;
```

Notice that we are using **goto** to jump out of a nested loop. Never jump into a loop or other structured statement, since this can have unpredictable effects.

Structured Statements

Structured statements are built from other statements. Use a structured statement when you want to execute other statements sequentially, conditionally, or repeatedly.

- A compound or **with** statement simply executes a sequence of constituent statements.
- A conditional statement that is an **if** or **case** statement executes at most one of its constituents, depending on specified criteria.
- Loop statements including **repeat**, **while**, and **for** loops execute a sequence of constituent statements repeatedly.
- A special group of statements including **raise**, **try**...**except**, and **try**...**finally** constructions create and handle exceptions. For information about exception generation and handling, see Exceptions (↗ see page 541).

Compound Statements

A compound statement is a sequence of other (simple or structured) statements to be executed in the order in which they are written. The compound statement is bracketed by the reserved words **begin** and **end**, and its constituent statements are separated by semicolons. For example:

```
begin
   Z := X;
   X := Y;
   X := Y;
          end;
```

The last semicolon before end is optional. So we could have written this as

```
begin
   Z := X;
   X := Y;
   Y := Z
end;
```

Compound statements are essential in contexts where Delphi syntax requires a single statement. In addition to program, function, and procedure blocks, they occur within other structured statements, such as conditionals or loops. For example:

```
begin
   I := SomeConstant;
   while I > 0 do
    begin
       ...
       I := I - 1;
    end;
end;
```

You can write a compound statement that contains only a single constituent statement; like parentheses in a complex term, **begin** and **end** sometimes serve to disambiguate and to improve readability. You can also use an empty compound statement to create a block that does nothing:

```
begin
end;
```

With Statements

A **with** statement is a shorthand for referencing the fields of a record or the fields, properties, and methods of an object. The syntax of a **with** statement is

1. **with***obj***do***statement*, or
2. **with***obj1, ..., objn***do***statement*

where *obj* is an expression yielding a reference to a record, object instance, class instance, interface or class type (metaclass) instance, and statement is any simple or structured statement. Within the *statement*, you can refer to fields, properties, and methods of *obj* using their identifiers alone, that is, without qualifiers.

For example, given the declarations

```
type
  TDate = record
    Day: Integer;
    Month: Integer;
    Year: Integer;
  end;

var
  OrderDate: TDate;
```

you could write the following **with** statement.

```
with OrderDate do
 if Month = 12 then
   begin
     Month := 1;
     Year := Year + 1;
   end
 else
   Month := Month + 1;
```

you could write the following **with** statement.

```
if OrderDate.Month = 12 then
  begin
   OrderDate.Month := 1;
   OrderDate.Year := OrderDate.Year + 1;
  end
else
  OrderDate.Month := OrderDate.Month + 1;
```

If the interpretation of *obj* involves indexing arrays or dereferencing pointers, these actions are performed once, before statement is executed. This makes **with** statements efficient as well as concise. It also means that assignments to a variable within statement cannot affect the interpretation of *obj* during the current execution of the **with** statement.

Each variable reference or method name in a **with** statement is interpreted, if possible, as a member of the specified object or record. If there is another variable or method of the same name that you want to access from the **with** statement, you need to prepend it with a qualifier, as in the following example.

```
with OrderDate do
  begin
    Year := Unit1.Year;
        ...
  end;
```

When multiple objects or records appear after **with**, the entire statement is treated like a series of nested **with** statements. Thus

with*obj1, obj2, ..., objn***do***statement*

is equivalent to

```
with obj1 do
  with obj2 do
    ...
    with objn do
      // statement
```

In this case, each variable reference or method name in statement is interpreted, if possible, as a member of *objn*; otherwise it is interpreted, if possible, as a member of *objn1*; and so forth. The same rule applies to interpreting the *objs* themselves, so that, for instance, if *objn* is a member of both *obj1* and *obj2*, it is interpreted as *obj2.objn*.

If Statements

There are two forms of **if** statement: **if**...**then** and the **if**...**then**...**else**. The syntax of an **if**...**then** statement is

if*expression***then***statement*

where *expression* returns a **Boolean** value. If expression is **True**, then *statement* is executed; otherwise it is not. For example,

```
if J <> 0 then Result := I / J;
```

The syntax of an **if**...**then**...**else** statement is

if*expression***then***statement1***else***statement2*

where *expression* returns a **Boolean** value. If expression is **True**, then *statement1* is executed; otherwise *statement2* is executed. For example,

```
if J = 0 then
   Exit
else
   Result := I / J;
```

The **then** and **else** clauses contain one statement each, but it can be a structured statement. For example,

```
if J <> o then
  begin
    Result := I / J;
    Count := Count + 1;
  end
else if Count = Last then
        Done := True
else
  Exit;
```

Notice that there is never a semicolon between the **then** clause and the word **else**. You can place a semicolon after an entire **if** statement to separate it from the next statement in its block, but the **then** and **else** clauses require nothing more than a space or carriage return between them. Placing a semicolon immediately before **else** (in an **if** statement) is a common programming error.

A special difficulty arises in connection with nested **if** statements. The problem arises because some **if** statements have **else** clauses while others do not, but the syntax for the two kinds of statement is otherwise the same. In a series of nested conditionals where there are fewer **else** clauses than **if** statements, it may not seem clear which **else** clauses are bound to which **if**s. Consider a statement of the form

if*expression1***thenif***expression2***then***statement1***else***statement2*;

There would appear to be two ways to parse this:

if*expression1* then [**if***expression2***then***statement1***else***statement2*];

if*expression1***then** [**if***expression2***then***statement1*] **else***statement2*;

The compiler always parses in the first way. That is, in real code, the statement

```
if ... { expression1} then
  if ... {expression2} then
    ... {statement1}
  else
    ... {statement2}
```

is equivalent to

```
if ... {expression1} then
  begin
    if ... {expression2} then
      ... {statement1}
    else
      ... {statement2}
end;
```

The rule is that nested conditionals are parsed starting from the innermost conditional, with each **else** bound to the nearest available **if** on its left. To force the compiler to read our example in the second way, you would have to write it explicitly as

```
if ... {expression1} then
```

```
  begin
   if ... {expression2} then
     ... {statement1}
   end
end
else
   ... {statement2};
```

Case Statements

The **case** statement may provide a readable alternative to deeply nested **if** conditionals. A **case** statement has the form

```
case selectorExpression of
   caseList1: statement1;
     ...
   caseListn: statementn;
end
```

where *selectorExpression* is any expression of an ordinal type smaller than 32 bits (string types and ordinals larger than 32 bits are invalid) and each *caseList* is one of the following:

- A numeral, declared constant, or other expression that the compiler can evaluate without executing your program. It must be of an ordinal type compatible with *selectorExpression*. Thus `7`, **True**, `4 + 5 * 3`, 'A', and `Integer('A')` can all be used as *caseLists*, but variables and most function calls cannot. (A few built-in functions like `Hi` and `Lo` can occur in a *caseList*. See Constant expressions (see page 589).)
- A subrange having the form *First..Last*, where *First* and *Last* both satisfy the criterion above and *First* is less than or equal to *Last*.
- A list having the form *item1, ..., itemn*, where each *item* satisfies one of the criteria above.

Each value represented by a *caseList* must be unique in the **case** statement; subranges and lists cannot overlap. A **case** statement can have a final **else** clause:

```
case selectorExpression of
   caseList1: statement1;
      ...
   caselistn: statementn;
   else
      statements;
end
```

where *statements* is a semicolon-delimited sequence of statements. When a **case** statement is executed, at most one of *statement1 ... statementn* is executed. Whichever *caseList* has a value equal to that of *selectorExpression* determines the statement to be used. If none of the *caseLists* has the same value as *selectorExpression*, then the statements in the **else** clause (if there is one) are executed.

The **case** statement

```
case I of
   1..5: Caption := 'Low';
   6..9: Caption := 'High';
   0, 10..99: Caption := 'Out of range';
   else
     Caption := '';
end
```

is equivalent to the nested conditional

```
if I in [1..5] then
   Caption := 'Low';
else if I in [6..10] then
        Caption := 'High';
     else if (I = 0) or (I in [10..99]) then
            Caption := 'Out of range'
```

```
          else
            Caption := '';
```

Other examples of **case** statements

```
case MyColor of
   Red: X := 1;
   Green: X := 2;
   Blue: X = 3;
   Yellow, Orange, Black: X := 0;
end;

case Selection of
   Done: Form1.Close;
   Compute: calculateTotal(UnitCost, Quantity);
   else
      Beep;
end;
```

Control Loops

Loops allow you to execute a sequence of statements repeatedly, using a control condition or variable to determine when the execution stops. Delphi has three kinds of control loop: **repeat** statements, **while** statements, and **for** statements.

You can use the standard `Break` and `Continue` procedures to control the flow of a **repeat**, **while**, or **for** statement. `Break` terminates the statement in which it occurs, while `Continue` begins executing the next iteration of the sequence.

Repeat Statements

The syntax of a **repeat** statement is

repeat*statement1; ...; statementn;***until***expression*

where *expression* returns a **Boolean** value. (The last semicolon before until is optional.) The **repeat** statement executes its sequence of constituent statements continually, testing *expression* after each iteration. When *expression* returns **True**, the **repeat** statement terminates. The sequence is always executed at least once because *expression* is not evaluated until after the first iteration.

Examples of **repeat** statements include

```
repeat
  K := I mod J;
  I := J;
  J := K;
until J = 0;

repeat
  Write('Enter a value (0..9): ');
  Readln(I);
until (I >= 0) and (I <= 9);
```

While Statements

A **while** statement is similar to a **repeat** statement, except that the control condition is evaluated before the first execution of the statement sequence. Hence, if the condition is false, the statement sequence is never executed.

The syntax of a **while** statement is

while*expression***do***statement*

where *expression* returns a **Boolean** value and *statement* can be a compound statement. The **while** statement executes its constituent *statement* repeatedly, testing *expression* before each iteration. As long as *expression* returns **True**, execution continues.

Examples of **while** statements include

```
while Data[I] <> X do I := I + 1;

  while I > 0 do
   begin
      if Odd(I) then Z := Z * X;
         I := I div 2;
         X := Sqr(X);
   end;

  while not Eof(InputFile) do
   begin
      Readln(InputFile, Line);
      Process(Line);
   end;
```

For Statements

A **for** statement, unlike a **repeat** or **while** statement, requires you to specify explicitly the number of iterations you want the loop to go through. The syntax of a **for** statement is

for *counter := initialValue* **to** *finalValue* **do** *statement*

or

for *counter := initialValue* **downto** *finalValue* **do** *statement*

where

- *counter* is a local variable (declared in the block containing the **for** statement) of ordinal type, without any qualifiers.
- *initialValue* and *finalValue* are expressions that are assignment-compatible with counter.
- *statement* is a simple or structured statement that does not change the value of counter.

The **for** statement assigns the value of *initialValue* to *counter*, then executes statement repeatedly, incrementing or decrementing *counter* after each iteration. (The **for**...**to** syntax increments *counter*, while the **for**...**downto** syntax decrements it.) When *counter* returns the same value as *finalValue*, *statement* is executed once more and the **for** statement terminates. In other words, *statement* is executed once for every value in the range from *initialValue* to *finalValue*. If *initialValue* is equal to *finalValue*, *statement* is executed exactly once. If *initialValue* is greater than *finalValue* in a **for**...**to** statement, or less than *finalValue* in a **for**...**downto** statement, then *statement* is never executed. After the **for** statement terminates (provided this was not forced by a `Break` or an `Exit` procedure), the value of *counter* is undefined.

Warning: The iteration variable *counter* cannot be modified within the loop. This includes assignment, and passing the variable to a var

parameter of a procedure. Doing so results in a compile-time warning. For purposes of controlling execution of the loop, the expressions *initialValue* and *finalValue* are evaluated only once, before the loop begins. Hence the **for**...**to** statement is almost, but not quite, equivalent to this **while** construction:

```
begin
   counter := initialValue;
   while counter <= finalValue do
    begin
      ... {statement};
      counter := Succ(counter);
    end;
end
```

The difference between this construction and the **for**...**to** statement is that the **while** loop reevaluates *finalValue* before each iteration. This can result in noticeably slower performance if *finalValue* is a complex expression, and it also means that changes to the value of *finalValue* within *statement* can affect execution of the loop.

Examples of **for** statements:

```
for I := 2 to 63 do
  if Data[I] > Max then
     Max := Data[I];

for I := ListBox1.Items.Count - 1 downto 0 do
    ListBox1.Items[I] := UpperCase(ListBox1.Items[I]);

for I := 1 to 10 do
    for J := 1 to 10 do
       begin
           X := 0;
           for K := 1 to 10 do
               X := X + Mat1[I,K] * Mat2[K,J];
           Mat[I,J] := X;
       end;

for C := Red to Blue do Check(C);
```

Iteration Over Containers Using For statements

Both Delphi for .NET and for Win32 support `for-element-in-collection` style iteration over containers. The following container iteration patterns are recognized by the compiler:

- `for Element in ArrayExpr do Stmt;`
- `for Element in StringExpr do Stmt;`
- `for Element in SetExpr do Stmt;`
- `for Element in CollectionExpr do Stmt;`
- `for Element in Record do Stmt;`

The type of the iteration variable `Element` must match the type held in the container. With each iteration of the loop, the iteration variable holds the current collection member. As with regular **for**-loops, the iteration variable must be declared within the same block as the **for** statement.

Warning: The iteration variable cannot be modified within the loop. This includes assignment, and passing the variable to a var

parameter of a procedure. Doing so results in a compile-time warning. Array expressions can be single or multidimensional, fixed length, or dynamic arrays. The array is traversed in increasing order, starting at the lowest array bound and ending at the array size minus one. The following code shows an example of traversing single, multi-dimensional, and dynamic arrays:

```
type
  TIntArray        = array[0..9] of Integer;
  TGenericIntArray = array of Integer;

var
  IArray1: array[0..9] of Integer   = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
  IArray2: array[1..10] of Integer  = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
  IArray3: array[1..2] of TIntArray = ((11, 12, 13, 14, 15, 16, 17, 18, 19, 20),
                                       (21, 22, 23, 24, 25, 26, 27, 28, 29, 30));
  MultiDimTemp: TIntArray;
  IDynArray:    TGenericIntArray;

  I: Integer;

begin

  for I in IArray1 do
     begin
```

```
          // Do something with I...
      end;

   // Indexing begins at lower array bound of 1.
   for I in IArray2 do
      begin
          // Do something with I...
      end;

   // Iterating a multi-dimensional array
   for MultiDimTemp in IArray3 do   // Indexing from 1..2
     for I in MultiDimTemp do       // Indexing from 0..9
       begin
           // Do something with I...
       end;

   // Iterating over a dynamic array
   IDynArray := IArray1;
   for I in IDynArray do
      begin
         // Do something with I...
      end;
```

The following example demonstrates iteration over string expressions:

```
var
  C: Char;
  S1, S2: String;
  Counter: Integer;

  OS1, OS2: ShortString;
  AC: AnsiChar;

begin

  S1 := 'Now is the time for all good men to come to the aid of their country.';
  S2 := '';

  for C in S1 do
    S2 := S2 + C;

  if S1 = S2 then
    WriteLn('SUCCESS #1');
  else
    WriteLn('FAIL #1');

  OS1 := 'When in the course of human events it becomes necessary to dissolve...';
  OS2 := '';

  for AC in OS1 do
    OS2 := OS2 + AC;

  if OS1 = OS2 then
    WriteLn('SUCCESS #2');
  else
    WriteLn('FAIL #2');

end.
```

The following example demonstrates iteration over set expressions:

```
type

  TMyThing = (one, two, three);
  TMySet   = set of TMyThing;
  TCharSet = set of Char;
```

```
var
  MySet:   TMySet;
  MyThing: TMyThing;

  CharSet: TCharSet;
  {$IF DEFINED(CLR)}
  C: AnsiChar;
  {$ELSE}
  C: Char;
  {$IFEND}

begin

  MySet := [one, two, three];
  for MyThing in MySet do
   begin
    // Do something with MyThing...
   end;


  CharSet := [#0..#255];
  for C in CharSet do
    begin
      // Do something with C...
    end;

end.
```

To use the **for**-**in** loop construct on a class or interface, the class or interface must implement a prescribed collection pattern. A type that implements the collection pattern must have the following attributes:

- The class or interface must contain a public instance method called `GetEnumerator()`. The `GetEnumerator()` method must return a class, interface, or record type.
- The class, interface, or record returned by `GetEnumerator()` must contain a public instance method called `MoveNext()`. The `MoveNext()` method must return a **Boolean**.
- The class, interface, or record returned by `GetEnumerator()` must contain a public instance, read-only property called `Current`. The type of the `Current` property must be the type contained in the collection.

If the enumerator type returned by `GetEnumerator()` implements the IDisposable interface, the compiler will call the Dispose method of the type when the loop terminates.

The following code demonstrates iterating over an enumerable container in Delphi.

```
type
  TMyIntArray = array of Integer;

  TMyEnumerator = class
    Values: TMyIntArray;
    Index:  Integer;
  public
    constructor Create;
    function GetCurrent: Integer;
    function MoveNext:   Boolean;
    property Current:    Integer read GetCurrent;
  end;

  TMyContainer  = class
  public
   function GetEnumerator: TMyEnumerator;
  end;

constructor TMyEnumerator.Create;
begin
  inherited Create;
  Values := TMyIntArray.Create(100, 200, 300);
  Index := -1;
```

3

```
end;

function TMyEnumerator.MoveNext: Boolean;
begin
  if Index < High(Values) then
    begin
      Inc(Index);
      Result := True;
    end
  else
    Result := False;
end;

function TMyEnumerator.GetCurrent: Integer;
begin
  Result := Values[Index];
end;

function TMyContainer.GetEnumerator: TMyEnumerator;
begin
  Result := TMyEnumerator.Create;
end;

var
  MyContainer: TMyContainer;
  I: Integer;

  Counter: Integer;

begin
  MyContainer := TMyContainer.Create;

  Counter := 0;
  for I in MyContainer do
    Inc(Counter, I);

  WriteLn('Counter = ', Counter);
end.
```

The following classes and their descendents support the **for**-**in** syntax:

- TList
- TCollection
- TStrings
- TInterfaceList
- TComponent
- TMenuItem
- TCustomActionList
- TFields
- TListItems
- TTreeNodes
- TToolBar

Blocks and Scope

Declarations and statements are organized into *blocks*, which define local namespaces (or *scopes*) for labels and identifiers. Blocks allow a single identifier, such as a variable name, to have different meanings in different parts of a program. Each block is part of the declaration of a program, function, or procedure; each program, function, or procedure declaration has one block.

Blocks

A block consists of a series of declarations followed by a compound statement. All declarations must occur together at the beginning of the block. So the form of a block is

```
{declarations}
begin
  {statements}
end
```

The *declarations* section can include, in any order, declarations for variables, constants (including resource strings), types, procedures, functions, and labels. In a program block, the *declarations* section can also include one or more **exports** clauses (see Libraries and packages (see page 635)).

For example, in a function declaration like

```
function UpperCase(const S: string): string;
var
  Ch: Char;
  L: Integer;
  Source, Dest: PChar;
begin
   ...
end;
```

the first line of the declaration is the function heading and all of the succeeding lines make up the block. `Ch`, `L`, `Source`, and `Dest` are local variables; their declarations apply only to the `UpperCase` function block and override, in this block only, any declarations of the same identifiers that may occur in the **program** block or in the **interface** or **implementation** section of a unit.

Scope

An identifier, such as a variable or function name, can be used only within the scope of its declaration. The location of a declaration determines its scope. An identifier declared within the declaration of a program, function, or procedure has a scope limited to the block in which it is declared. An identifier declared in the interface section of a unit has a scope that includes any other units or programs that use the unit where the declaration occurs. Identifiers with narrower scope, especially identifiers declared in functions and procedures, are sometimes called local, while identifiers with wider scope are called global.

The rules that determine identifier scope are summarized below.

If the identifier is declared in ...	its scope extends ...
the declaration section of a program, function, or procedure	from the point where it is declared to the end of the current block, including all blocks enclosed within that scope.
the interface section of a unit	from the point where it is declared to the end of the unit, and to any other unit or program that uses that unit. (See Programs and Units (see page 683).)
the implementation section of a unit, but not within the block of any function or procedure	from the point where it is declared to the end of the unit. The identifier is available to any function or procedure in the unit, including the initialization and finalization sections, if present.
the definition of a record type (that is, the identifier is the name of a field in the record)	from the point of its declaration to the end of the record-type definition. (See Records (see page 566).)
the definition of a class (that is, the identifier is the name of a data field property or method in the class)	from the point of its declaration to the end of the class-type definition, and also includes descendants of the class and the blocks of all methods in the class and its descendants. (See Classes and Objects (see page 514).)

Naming Conflicts

When one block encloses another, the former is called the outer block and the latter the inner block. If an identifier declared in an outer block is redeclared in an inner block, the inner declaration takes precedence over the outer one and determines the meaning of the identifier for the duration of the inner block. For example, if you declare a variable called `MaxValue` in the **interface** section of a unit, and then declare another variable with the same name in a function declaration within that unit, any unqualified occurrences of `MaxValue` in the function block are governed by the second, local declaration. Similarly, a function declared within another function creates a new, inner scope in which identifiers used by the outer function can be redeclared locally.

The use of multiple units further complicates the definition of scope. Each unit listed in a **uses** clause imposes a new scope that encloses the remaining units used and the **program** or **unit** containing the **uses** clause. The first unit in a **uses** clause represents the outermost scope and each succeeding unit represents a new scope inside the previous one. If two or more units declare the same identifier in their **interface** sections, an unqualified reference to the identifier selects the declaration in the innermost scope, that is, in the unit where the reference itself occurs, or, if that unit doesn't declare the identifier, in the last unit in the `uses` clause that does declare the identifier.

The `System` and `SysInit` units are used automatically by every program or unit. The declarations in `System`, along with the predefined types, routines, and constants that the compiler understands automatically, always have the outermost scope.

You can override these rules of scope and bypass an inner declaration by using a qualified identifier (see Qualified Identifiers (see page 701)) or a **with** statement (see With Statements, above).

See Also

Fundamental Syntactic Elements (see page 701)

Expressions (see page 720)

3.1.3.14.3 Expressions

circuit evaluation is usually preferable because it guarantees minimum execution time and, in most cases, minimum code size. Complete evaluation is sometimes convenient when one operand is a function with side effects that alter the execution of the program.

circuit evaluation also allows the use of constructions that might otherwise result in illegal runtime operations. For example, the following code iterates through the string `S`, up to the first comma.

```
while (I <= Length(S)) and (S[I] <> ',') do
             begin
                  ...
                  Inc(I);
             end;
```

In the case where `S` has no commas, the last iteration increments `I` to a value which is greater than the length of `S`. When the **while** condition is next tested, complete evaluation results in an attempt to read `S[I]`, which could cause a runtime error. Under short-circuit evaluation, in contrast, the second part of the **while** condition `(S[I] <> ',')` is not evaluated after the first part fails.

Use the `$B` compiler directive to control evaluation mode. The default state is `{$B}`, which enables short-circuit evaluation. To enable complete evaluation locally, add the `{$B+}` directive to your code. You can also switch to complete evaluation on a project-wide basis by selecting **Complete Boolean Evaluation** in the **Compiler Options** dialog (all source units will need to be recompiled).

Note: If either operand involves a Variant, the compiler always performs complete evaluation (even in the `{$B}` state).

Logical (Bitwise) Operators

The following logical operators perform bitwise manipulation on integer operands. For example, if the value stored in `X` (in binary) is `001101` and the value stored in `Y` is `100001`, the statement:

```
Z := X or Y;
```

assigns the value `101101` to `Z`.

Logical (Bitwise) Operators

Operator	Operation	Operand Types	Result Type	Example
not	bitwise negation	integer	integer	`not X`
and	bitwise and	integer	integer	`X and Y`
or	bitwise or	integer	integer	`X or Y`
xor	bitwise xor	integer	integer	`X xor Y`
shl	bitwise shift left	integer	integer	`X shl 2`
shr	bitwise shift right	integer	integer	`Y shr I`

The following rules apply to bitwise operators.

- The result of a **not** operation is of the same type as the operand.
- If the operands of an **and**, **or**, or **xor** operation are both integers, the result is of the predefined integer type with the smallest range that includes all possible values of both types.
- The operations `x shl y` and `x shr y` shift the value of `x` to the left or right by `y` bits, which (if `x` is an unsigned integer) is equivalent to multiplying or dividing `x` by `2^y`; the result is of the same type as `x`. For example, if `N` stores the value `01101` (decimal 13), then `N sh 1` returns `11010` (decimal 26). Note that the value of `y` is interpreted modulo the size of the type of `x`. Thus for example, if `x` is an integer, `x shl 40` is interpreted as `x shl 8` because an integer is 32 bits and 40 mod 32 is 8.

String Operators

The relational operators **=**, **<>**, **<**, **>**, **<=**, and **>=** all take string operands (see Relational operators). The **+** operator concatenates two strings.

String Operators

Operator	Operation	Operand Types	Result Type	Example
+	concatenation	string, packed string, character	string	`S + '. '`

The following rules apply to string concatenation.

- The operands for **+** can be strings, packed strings (packed arrays of type **Char**), or characters. However, if one operand is of type **WideChar**, the other operand must be a long string (**AnsiString** or **WideString**).
- The result of a **+** operation is compatible with any string type. However, if the operands are both short strings or characters, and their combined length is greater than 255, the result is truncated to the first 255 characters.

Pointer Operators

The relational operators **<**, **>**, **<=**, and **>=** can take operands of type **PChar** and **PWideChar** (see Relational operators). The following operators also take pointers as operands. For more information about pointers, see Pointers and pointer types (↗ see page 575).

Character-pointer operators

Operator	Operation	Operand Types	Result Type	Example
+	pointer addition	character pointer, integer	character pointer	`P + I`
-	pointer subtraction	character pointer, integer	character pointer, integer	`P - Q`
^	pointer dereference	pointer	base type of pointer	`P^`
=	equality	pointer	Boolean	`P = Q`
<>	inequality	pointer	Boolean	`P <> Q`

The **^** operator dereferences a pointer. Its operand can be a pointer of any type except the generic **Pointer**, which must be typecast before dereferencing.

`P = Q` is **True** just in case `P` and `Q` point to the same address; otherwise, `P <> Q` is **True**.

You can use the **+** and **-** operators to increment and decrement the offset of a character pointer. You can also use **-** to calculate the difference between the offsets of two character pointers. The following rules apply.

- If `I` is an integer and `P` is a character pointer, then `P + I` adds `I` to the address given by `P`; that is, it returns a pointer to the address `I` characters after `P`. (The expression `I + P` is equivalent to `P + I`.) `P - I` subtracts `I` from the address given by `P`; that is, it returns a pointer to the address `I` characters before `P`. This is true for **PChar** pointers; for **PWideChar** pointers `P + I` adds `SizeOf(WideChar)` to `P`.
- If `P` and `Q` are both character pointers, then `P - Q` computes the difference between the address given by `P` (the higher address) and the address given by `Q` (the lower address); that is, it returns an integer denoting the number of characters between `P` and `Q`. `P + Q` is not defined.

Set Operators

The following operators take sets as operands.

Set Operators

Operator	Operation	Operand Types	Result Type	Example
+	union	set	set	`Set1 + Set2`
-	difference	set	set	`S - T`
*****	intersection	set	set	`S * T`
<=	subset	set	Boolean	`Q <= MySet`
>=	superset	set	Boolean	`S1 >= S2`
=	equality	set	Boolean	`S2 = MySet`
<>	inequality	set	Boolean	`MySet <> S1`
in	membership	ordinal, set	Boolean	`A in Set1`

The following rules apply to **+**, **-**, and *****.

- An ordinal `O` is in `X + Y` if and only if `O` is in `X` or `Y` (or both). `O` is in `X - Y` if and only if `O` is in `X` but not in `Y`. `O` is in `X * Y` if and only if `O` is in both `X` and `Y`.
- The result of a **+**, **-**, or ***** operation is of the type `set of A..B`, where `A` is the smallest ordinal value in the result set and `B` is the largest.

The following rules apply to **<=**, **>=**, **=**, **<>**, and **in**.

- `X <= Y` is **True** just in case every member of `X` is a member of `Y`; `Z >= W` is equivalent to `W <= Z`. `U = V` is **True** just in case `U` and `V` contain exactly the same members; otherwise, `U <> V` is **True**.
- For an ordinal `O` and a set `S`, `O in S` is **True** just in case `O` is a member of `S`.

Relational Operators

Relational operators are used to compare two operands. The operators **=**, **<>**, **<=**, and **>=** also apply to sets.

Relational Operators

Operator	Operation	Operand Types	Result Type	Example
=	equality	simple, class, class reference, interface, string, packed string	Boolean	`I = Max`
<>	inequality	simple, class, class reference, interface, string, packed string	Boolean	`X <> Y`
<	less-than	simple, string, packed string, PChar	Boolean	`X < Y`
>	greater-than	simple, string, packed string, PChar	Boolean	`Len > 0`
<=	less-than-or-equal-to	simple, string, packed string, PChar	Boolean	`Cnt <= I`
>=	greater-than-or-equal-to	simple, string, packed string, PChar	Boolean	`I >= 1`

For most simple types, comparison is straightforward. For example, `I = J` is **True** just in case `I` and `J` have the same value, and `I <> J` is **True** otherwise. The following rules apply to relational operators.

- Operands must be of compatible types, except that a real and an integer can be compared.
- Strings are compared according to the ordinal values that make up the characters that make up the string. Character types are treated as strings of length 1.
- Two packed strings must have the same number of components to be compared. When a packed string with `n` components is compared to a string, the packed string is treated as a string of length `n`.
- Use the operators **<**, **>**, **<=**, and **>=** to compare **PChar** (and **PWideChar**) operands only if the two pointers point within the same character array.
- The operators **=** and **<>** can take operands of class and class-reference types. With operands of a class type, **=** and **<>** are evaluated according the rules that apply to pointers: `C = D` is **True** just in case `C` and `D` point to the same instance object, and `C <> D` is **True** otherwise. With operands of a class-reference type, `C = D` is **True** just in case `C` and `D` denote the same class, and `C <> D` is **True** otherwise. This does not compare the data stored in the classes. For more information about classes, see Classes and objects (↗ see page 514).

Class Operators

The operators **as** and **is** take classes and instance objects as operands; **as** operates on interfaces as well. For more information, see Classes and objects (↗ see page 514) and Object interfaces (↗ see page 625).

The relational operators **=** and **<>** also operate on classes.

The @ Operator

The @ operator returns the address of a variable, or of a function, procedure, or method; that is, @ constructs a pointer to its operand. For more information about pointers, see Pointers and pointer types (↗ see page 575). The following rules apply to @.

- If `X` is a variable, `@X` returns the address of `X`. (Special rules apply when `X` is a procedural variable; see Procedural types in statements and expressions (see page 578).) The type of @X is **Pointer** if the default {$T} compiler directive is in effect. In the {$T+} state, @X is of type ^T, where T is the type of X (this distinction is important for assignment compatibility, see Assignment-compatibility).

- If F is a routine (a function or procedure), @F returns F's entry point. The type of @F is always **Pointer**.
- When @ is applied to a method defined in a class, the method identifier must be qualified with the class name. For example,

```
@TMyClass.DoSomething
```

points to the DoSomething method of TMyClass. For more information about classes and methods, see Classes and objects (see page 514).

Note: When using the @ operator, it is not possible to take the address of an interface method as the address is not known at compile time and cannot be extracted at runtime.

Operator Precedence

In complex expressions, rules of precedence determine the order in which operations are performed.

Precedence of operators

Operators	Precedence
@, **not**	first (highest)
*****, **/**, **div**, **mod**, **and**, **shl**, **shr**, **as**	second
+, **-**, **or**, **xor**	third
=, **<>**, **<**, **>**, **<=**, **>=**, **in**, **is**	fourth (lowest)

An operator with higher precedence is evaluated before an operator with lower precedence, while operators of equal precedence associate to the left. Hence the expression

```
X + Y * Z
```

multiplies Y times Z, then adds X to the result; ***** is performed first, because is has a higher precedence than **+**. But

```
X - Y + Z
```

first subtracts Y from X, then adds Z to the result; **-** and **+** have the same precedence, so the operation on the left is performed first.

You can use parentheses to override these precedence rules. An expression within parentheses is evaluated first, then treated as a single operand. For example,

```
(X + Y) * Z
```

multiplies Z times the sum of X and Y.

Parentheses are sometimes needed in situations where, at first glance, they seem not to be. For example, consider the expression

```
X = Y or X = Z
```

The intended interpretation of this is obviously

```
(X = Y) or (X = Z)
```

Without parentheses, however, the compiler follows operator precedence rules and reads it as

```
(X = (Y or X)) = Z
```

which results in a compilation error unless Z is Boolean.

Parentheses often make code easier to write and to read, even when they are, strictly speaking, superfluous. Thus the first example could be written as

```
X + (Y * Z)
```

Here the parentheses are unnecessary (to the compiler), but they spare both programmer and reader from having to think about operator precedence.

Description

This topic describes syntax rules of forming Delphi expressions.

The following items are covered in this topic:

- Valid Delphi Expressions
- Operators
- Function calls
- Set constructors
- Indexes
- Typecasts

Expressions

An expression is a construction that returns a value. The following table shows examples of Delphi expressions:

`X`	variable
`@X`	address of the variable X
`15`	integer constant
`InterestRate`	variable
`Calc(X, Y)`	function call
`X * Y`	product of X and Y
`Z / (1 - Z)`	quotient of Z and (1 - Z)
`X = 1.5`	Boolean
`C in Range1`	Boolean
`not Done`	negation of a Boolean
`['a', 'b', 'c']`	set
`Char(48)`	value typecast

The simplest expressions are variables and constants (described in Data types (see page 553)). More complex expressions are built from simpler ones using operators, function calls, set constructors, indexes, and typecasts.

Operators

Operators behave like predefined functions that are part of the Delphi language. For example, the expression `(X + Y)` is built from the variables `X` and `Y`, called operands, with the **+** operator; when `X` and `Y` represent integers or reals, `(X + Y)` returns their sum. Operators include @, **not**, **^**, *, **/**, **div**, **mod**, **and**, **shl**, **shr**, **as**, **+**, **-**, **or**, **xor**, **=**, **>**, **<**, **<>**, **<=**, **>=**, **in**, and **is**.

The operators @, **not**, and **^** are unary (taking one operand). All other operators are binary (taking two operands), except that **+** and **-** can function as either a unary or binary operator. A unary operator always precedes its operand (for example, `-B`), except for **^**, which follows its operand (for example, `P^`). A binary operator is placed between its operands (for example, `A = 7`).

Some operators behave differently depending on the type of data passed to them. For example, **not** performs bitwise negation on an integer operand and logical negation on a **Boolean** operand. Such operators appear below under multiple categories.

Except for **^**, **is**, and **in**, all operators can take operands of type Variant (see page 580).

The sections that follow assume some familiarity with Delphi data types (see page 553).

For information about operator precedence in complex expressions, see Operator Precedence Rules, later in this topic.

Arithmetic Operators

Arithmetic operators, which take real or integer operands, include **+**, **-**, *****, **/**, **div**, and **mod**.

Binary Arithmetic Operators

Operator	Operation	Operand Types	Result Type	Example
+	addition	integer, real	integer, real	`X + Y`
-	subtraction	integer, real	integer, real	`Result - 1`
*	multiplication	integer, real	integer, real	`P * InterestRate`
/	real division	integer, real	real	`X / 2`
div	integer division	integer	integer	`Total div UnitSize`
mod	remainder	integer	integer	`Y mod 6`

Unary arithmetic operators

Operator	Operation	Operand Type	Result Type	Example
+	sign identity	integer, real	integer, real	`+7`
-	sign negation	integer, real	integer, real	`-X`

The following rules apply to arithmetic operators.

- The value of `x / y` is of type **Extended**, regardless of the types of `x` and `y`. For other arithmetic operators, the result is of type **Extended** whenever at least one operand is a real; otherwise, the result is of type **Int64** when at least one operand is of type **Int64**; otherwise, the result is of type **Integer**. If an operand's type is a subrange of an integer type, it is treated as if it were of the integer type.
- The value of `x div y` is the value of `x / y` rounded in the direction of zero to the nearest integer.
- The **mod** operator returns the remainder obtained by dividing its operands. In other words, `x mod y = x —(x div y) * y`.
- A runtime error occurs when y is zero in an expression of the form `x / y`, `x div y`, or `x mod y`.

Boolean Operators

The Boolean operators **not**, **and**, **or**, and **xor** take operands of any Boolean type and return a value of type **Boolean**.

Boolean Operators

Operator	Operation	Operand Types	Result Type	Example
not	negation	**Boolean**	**Boolean**	`not (C in MySet)`
and	conjunction	**Boolean**	**Boolean**	`Done and (Total > 0)`
or	disjunction	**Boolean**	**Boolean**	`A or B`
xor	exclusive disjunction	**Boolean**	**Boolean**	`A xor B`

These operations are governed by standard rules of Boolean logic. For example, an expression of the form `x and y` is **True** if and only if both `x` and `y` are **True**.

Complete Versus Short-Circuit Boolean Evaluation

The compiler supports two modes of evaluation for the **and** and **or** operators: complete evaluation and short-circuit (partial) evaluation. Complete evaluation means that each conjunct or disjunct is evaluated, even when the result of the entire expression is already determined. Short-circuit evaluation means strict left-to-right evaluation that stops as soon as the result of the entire expression is determined. For example, if the expression `A and B` is evaluated under short-circuit mode when `A` is **False**, the compiler won't evaluate `B`; it knows that the entire expression is **False** as soon as it evaluates `A`.

Function Calls

Because functions return a value, function calls are expressions. For example, if you've defined a function called `Calc` that takes two integer arguments and returns an integer, then the function call `Calc(24,47)` is an integer expression. If `I` and `J` are integer variables, then `I + Calc(J,8)` is also an integer expression. Examples of function calls include

```
Sum(A, 63)
Maximum(147, J)
Sin(X + Y)
Eof(F)
Volume(Radius, Height)
GetValue
TSomeObject.SomeMethod(I,J);
```

For more information about functions, see Procedures and functions (⧉ see page 662).

Set Constructors

A set constructor denotes a set-type value. For example,

```
[5, 6, 7, 8]
```

denotes the set whose members are 5, 6, 7, and 8. The set constructor

```
[ 5..8 ]
```

could also denote the same set.

The syntax for a set constructor is

[item1, ..., itemn]

where each item is either an expression denoting an ordinal of the set's base type or a pair of such expressions with two dots (**..**) in between. When an item has the form `x..y`, it is shorthand for all the ordinals in the range from `x` to `y`, including `y`; but if `x` is greater than `y`, then `x..y`, the set `[x..y]`, denotes nothing and is the empty set. The set constructor `[ ]` denotes the empty set, while `[x]` denotes the set whose only member is the value of `x`.

Examples of set constructors:

```
[red, green, MyColor]
[1, 5, 10..K mod 12, 23]
['A'..'Z', 'a'..'z', Chr(Digit + 48)]
```

For more information about sets, see Sets (⧉ see page 566).

Indexes

Strings, arrays, array properties, and pointers to strings or arrays can be indexed. For example, if `FileName` is a string variable, the expression `FileName[3]` returns the third character in the string denoted by `FileName`, while `FileName[I + 1]` returns the character immediately after the one indexed by `I`. For information about strings, see String types (⧉ see page 561). For information about arrays and array properties, see Arrays (⧉ see page 566) and Array properties (⧉ see page 530).

Typecasts

It is sometimes useful to treat an expression as if it belonged to different type. A typecast allows you to do this by, in effect, temporarily changing an expression's type. For example, `Integer('A')` casts the character A as an integer.

The syntax for a typecast is

typeIdentifier(expression)

If the expression is a variable, the result is called a variable typecast; otherwise, the result is a value typecast. While their syntax is the same, different rules apply to the two kinds of typecast.

Value Typecasts

In a value typecast, the type identifier and the cast expression must both be ordinal or pointer types. Examples of value typecasts include

```
Integer('A')
Char(48)
Boolean(0)
Color(2)
Longint(@Buffer)
```

The resulting value is obtained by converting the expression in parentheses. This may involve truncation or extension if the size of the specified type differs from that of the expression. The expression's sign is always preserved.

The statement

```
I := Integer('A');
```

assigns the value of `Integer('A')`, which is 65, to the variable `I`.

A value typecast cannot be followed by qualifiers and cannot appear on the left side of an assignment statement.

Variable Typecasts

You can cast any variable to any type, provided their sizes are the same and you do not mix integers with reals. (To convert numeric types, rely on standard functions like `Int` and `Trunc`.) Examples of variable typecasts include

```
Char(I)
Boolean(Count)
TSomeDefinedType(MyVariable)
```

Variable typecasts can appear on either side of an assignment statement. Thus

```
var MyChar: char;
  ...
  Shortint(MyChar) := 122;
```

assigns the character z (ASCII 122) to `MyChar`.

You can cast variables to a procedural type. For example, given the declarations

```
type Func = function(X: Integer): Integer;
var
  F: Func;
  P: Pointer;
  N: Integer;
```

you can make the following assignments.

```
F := Func(P);     { Assign procedural value in P to F }
Func(P) := F;     { Assign procedural value in F to P }
@F := P;          { Assign pointer value in P to F }
P := @F;          { Assign pointer value in F to P }
N := F(N);        { Call function via F }
N := Func(P)(N);  { Call function via P }
```

Variable typecasts can also be followed by qualifiers, as illustrated in the following example.

```
type
  TByteRec = record
     Lo, Hi: Byte;
  end;

  TWordRec = record
     Low, High: Word;
  end;

var
  B: Byte;
  W: Word;
  L: Longint;
  P: Pointer;

begin
  W := $1234;
  B := TByteRec(W).Lo;
  TByteRec(W).Hi := 0;
  L := $1234567;
  W := TWordRec(L).Low;
  B := TByteRec(TWordRec(L).Low).Hi;
  B := PByte(L)^;
end;
```

In this example, `TByteRec` is used to access the low- and high-order bytes of a word, and `TWordRec` to access the low- and high-order words of a long integer. You could call the predefined functions `Lo` and `Hi` for the same purpose, but a variable typecast has the advantage that it can be used on the left side of an assignment statement.

For information about typecasting pointers, see Pointers and pointer types (see page 575). For information about casting class and interface types, see The as operator (see page 539) and Interface typecasts (see page 631).

See Also

Fundamental Syntactic Elements (see page 701)

Declarations and Statements (see page 705)

3.2 RAD Studio Dialogs and Commands

This section contains help for dialogs and menu commands in the RAD Studio user interface.

Topics

Name	Description
Code Visualization (↗ see page 730)	
Components (↗ see page 732)	
Database (↗ see page 737)	
Edit (↗ see page 754)	
Error Messages (↗ see page 760)	
File (↗ see page 768)	
HTML Elements (↗ see page 801)	
Insert (↗ see page 813)	
Testing Wizards (↗ see page 815)	
NET_VS (↗ see page 817)	
Project (↗ see page 819)	
Propeditors (↗ see page 910)	
Run (↗ see page 936)	
Search (↗ see page 952)	
Together (↗ see page 957)	
Tools (↗ see page 979)	
View (↗ see page 1014)	
Win View (↗ see page 1062)	

3.2.1 Code Visualization

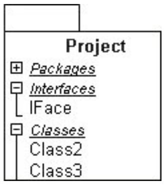
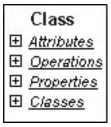
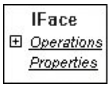
Topics

Name	Description
Code Visualization Diagram (↗ see page 730)	The Code Visualization diagram displays your project as a UML static structure diagram. The following table shows the standard UML notations used on the diagram.
Export Diagram to Image (↗ see page 731)	**Export to Image...** Use this dialog to create a file with the **Code Visualization diagram**. Various graphics file formats are supported, including BMP, JPG, and GIF.

3.2.1.1 Code Visualization Diagram

The Code Visualization diagram displays your project as a UML static structure diagram.

The following table shows the standard UML notations used on the diagram.

UML Notation	Represents
 Project Packages Interfaces IFace Classes Class2 Class3	A UML package
 	A class
 	An interface
+	A member with public visibility
#	A member with protected visibility
–	A member with private visibility
————▷	A generalization link
------▷	A realization link
------>	A dependency link
————	An association link

Tip: If the diagram is too large to view all at once, you can use the Overview

window to scroll to that portion you wish to view.

See Also

Using Code Visualization

Using the Model View Window and Code Visualization Diagram

Using the Overview Window

3.2.1.2 Export Diagram to Image

Export to Image...

Use this dialog to create a file with the **Code Visualization diagram**. Various graphics file formats are supported, including BMP, JPG, and GIF.

Item	Description
Z	The zoom (magnification) factor for the image. You can enter a numeric value in this field, or use the **Preview zoom** slider on the preview pane. A value of 1.0 creates a full size image, while (for example) a value of 0.5 scales the image to half its size. A value of 1.5 scales the image to 1.5 times its size.
W	The width of the image. The image will be scaled to the new size, and the zoom factor field will be updated to reflect the new scaling value.
H	The height of the image. The image will be scaled to the new size, and the zoom factor field will be updated to reflect the new scaling value.

Preview	Click to show or hide a preview of the image. The preview shows how the image will look, given the current values of Z, W, and H.
Preview zoom	Use the slider control to scale the image to a smaller size. When the slider is at the leftmost position, a full size image (1:1) will be created. When at the rightmost position, the image size will be decreased by 16 times (1:16).
Auto preview zoom	Click to have the image **Preview pane** update in real-time, as you slide the **Preview zoom** control.
Save	Click to open a Windows Save-as dialog that will let you choose the graphical file format for the image file.

See Also

Using Code Visualization

Using the Model View Window and Code Visualization Diagram

Using the Overview Window

3.2.2 Components

Topics

Name	Description
Create Component Template (↗ see page 732)	**Component ▶ Create Component Template** Use this dialog box to save the selected components on the current form as a reusable component template.
Import Component (↗ see page 733)	**Component ▶ Import Component** Imports a type library, ActiveX Control, or a .NET assembly. You need to choose **VCL for C++** or **VCL for Delphi** if you select Import Component with no project open.
Packages (↗ see page 734)	**Components ▶ Install Packages** Specifies the design time packages installed in the IDE and the runtime packages that you want to install on your system, for use on all projects.
Assembly Search Paths (↗ see page 734)	**Component ▶ Installed .NET Components** Use this page to change the assembly search path that RAD Studio uses when locating assemblies.
Installed .NET Components (↗ see page 734)	**Component ▶ Installed .NET Components** Controls which .NET components are displayed in the **Tool Palette**.
.NET VCL Components (↗ see page 735)	**Component ▶ Installed .NET Components** Use this page to control which .NET VCL components are displayed in the **Tool Palette**.
New Component (↗ see page 735)	**Component ▶ New VCL Component** Create the basic unit for a new component. You need to choose **VCL for C++**, **VCL for Delphi** or **VCL for Delphi .NET** if you select New VCL Component with no project open.
New VCL Component Wizard (↗ see page 736)	**Component ▶ New VCL Component** Use this wizard to create a new VCL.NET or VCL Win32 component.

3.2.2.1 Create Component Template

Component ▶ Create Component Template

Use this dialog box to save the selected components on the current form as a reusable component template.

Item	Description
Component name	Indicates the name of the new component template. By default, the name of the first component that you selected is displayed and the word Template is appended to it. You can change this name, but be careful not to duplicate existing component names.
Palette page	Select the **Tool Palette** category in which you want the new template to appear.
Palette icon	Indicates the icon used to represent the component template in the **Tool Palette**. By default, the icon of the first component you selected is displayed. To change it, click the **Change** button and choose a new file. The bitmap must be no larger than 24 pixels by 24 pixels.

See Also

Creating a Component Template (↗ see page 154)

3.2.2.2 Import Component

Component ▶ Import Component

Imports a type library, ActiveX Control, or a .NET assembly.

You need to choose **VCL for C++** or **VCL for Delphi** if you select Import Component with no project open.

Item	Description
Import a Type Library	Click to import a type library.
Import ActiveX Control	Click to import an ActiveX Control.
Import .NET assembly	Click to import a .NET assembly as a COM object.

Registered Type Libraries/ActiveX Controls/.NET Assemblies

Depending on the type of component selected in the previous step, this dialog will show a list of registered type libraries, ActiveX Controls, or .NET assemblies in the Global Assembly Cache.

Item	Description
Component list	Displays a list of registered type libraries, ActiveX Controls, or .NET assemblies in the Global Assembly Cache.
Add	Click to browse for a new component to add to the list.

Component

Use the component page to edit parameters used to import the component into the IDE.

Item	Description
Class Name	Displays the name of the class (or classes) that will be created.
Palette Page	Select the **Tool Palette** category where you want the components to appear.
Unit name	Type the name of the unit that will contain the component.
Browse button	Click to navigate to the folder where the unit file will reside.
Search path	Enter the search path for the new component. The default value is the environment options library search path. The folder designated in the unit name field will be appended to the search path.

3.2.2.3 Packages

Components ▶ Install Packages

Specifies the design time packages installed in the IDE and the runtime packages that you want to install on your system, for use on all projects.

Item	Description
Design packages	Lists the design time packages available to the IDE and all projects. Check the design packages you want to make available on a system-wide level. You do not need to have a project open to install packages on a system-wide level.
Add	Installs a design time package. The package will be available in all projects.
Remove	Deletes the selected package. The package becomes unavailable in all projects.
Edit	Opens the selected package in the Package Editor if the source code or .dcp file is available.

3.2.2.4 Assembly Search Paths

Component ▶ Installed .NET Components

Use this page to change the assembly search path that RAD Studio uses when locating assemblies.

Item	Description
Search Paths	Lists the paths that will be searched for assemblies when a component is created in the development environment. If a component exists in more than one assembly, it will be used from the first assembly in which it is found. Use the up and down arrow buttons to the right of the search path list to move the selected path up or down in the search order.
Replace	Replaces the currently selected path in the **Search Path** list with the path in the box above the **Replace** button.
Add	Adds the path displayed in the box above the **Add** button to the **Search Path** list. Use the browse button to navigate to an existing folder.
Delete	Deletes the currently selected path from the **Search Path** list.
Reset	Sets assembly path search paths back to the original configuration.

Tip: Click any column heading to sort the display.

3.2.2.5 Installed .NET Components

Component ▶ Installed .NET Components

Controls which .NET components are displayed in the **Tool Palette**.

Item	Description
☑	Indicates whether the component is displayed in the **Tool Palette**. Checked components are displayed; unchecked components are not displayed.
Name	The name of the component.

Category	The functional category to which the component belongs. At least one component name must be checked for the category to appear in the **Tool Palette**.
Namespace	The namespace to which the component belongs.
Assembly Name	The name of the assembly (.dll) that contains the component.
Assembly Path	The location of the assembly.
Category	Enter the category to which you want to add an assembly.
Select an Assembly	Displays a dialog box allowing you to browse to an assembly or executable file.
Reset	Sets installed components to the original configuration.

Tip: Click any column heading to sort the display.

3.2.2.6 .NET VCL Components

Component ▶Installed .NET Components

Use this page to control which .NET VCL components are displayed in the **Tool Palette**.

Item	Description
☑	Indicates whether this category of components is displayed in the **Tool Palette**.
Name	The name of the component.
Namespace/unit	The namespace and unit to which the component belongs.
Add	Displays a dialog box, allowing you to navigate to and install a DLL containing .NET VCL components.
Remove	Removes the selected (highlighted) category of components from the **Tool Palette**.
Reset	Restores the original configuration of installed components.

Tip: The status bar at the bottom of this dialog displays the path for the selected component category.

3.2.2.7 New Component

Component ▶New VCL Component

Create the basic unit for a new component.

You need to choose **VCL for C++**, **VCL for Delphi** or **VCL for Delphi .NET** if you select New VCL Component with no project open.

Item	Description
Ancestor type	Use the drop-down list to select a base class, or enter the name of a base class for your new component. Unless you override them in the component declaration, your new component will inherit all the properties, methods, and events from its ancestor class. After you enter a base class, default entries are written to the **Class name** and **Unit file** name. You can accept or edit these entries.
Class name	The name of the new class you are creating. In general, classe names are prefaced with a T. For example, the name of your new button component might be TMyButton.
Palette page	Use the drop-down list to select a category, or enter the name of the category, in which you want your new component to appear on the **Tool Palette**.

Unit file name	Specifies the name of the unit that will contain the new component. You can include a directory path with the name; otherwise, the unit will be created in the current directory. If the unit directory is not in the **Search path**, it will be added to the end of the path.
Search path	Specifies the path the product uses to search for files.
OK	Creates the component but does not install it. To install the component later, choose **Component ▶ Installed .NET Components**.

3.2.2.8 New VCL Component Wizard

Component ▶ New VCL Component

Use this wizard to create a new VCL.NET or VCL Win32 component.

Item	Description
Delphi for VCL.NET	Click to create a new component for the VCL.NET framework and platform.
Delphi for VCL Win32	Click to create a new component for the VCL Win32 framework and platform.

Ancestor Component

The **Ancestor Component** page displays a list of installed components that can be used as ancestors for the new component. This dialog displays installed components for the platform selected on the previous page.

Item	Description
Component list	Shows a list of installed components for the platform selected in the previous page.

Component

The **Component** page allows you to edit parameters used to create the new component.

Item	Description
Class Name	Enter a class name for the new component, or accept the default value.
Palette Page	Select the **Tool Palette** category on which the new component will appear.
Unit name	Enter the name of the unit for the new component.
Search path	Enter the search path for the new component. The default value is the environment options library search path.

Install

Use the **Install** page to select an action for the IDE to take when it installs the new component.

Item	Description
Create a unit	Click to cause the IDE to create a new unit.
Install to Existing Package	Click to install the new component into a package that already exists.
Install to New Package	Click to cause the IDE to generate a new package.

Note: If the new component's platform is Win32 and the active project is a Win32 package, an additional radio button will appear allowing you to install the component into the active project. Similarly, if the new component's platform is .NET, and the active project is a .NET package, an additional radio button will appear allowing you to install the component into the active project.

Existing Package

Use this dialog to select an existing package in which to install the new component.

Item	Description
Installed package list	Lists all the packages currently installed on the system.

New Package

Use this dialog to enter the name of a new package in which to install the new component

Item	Description
File name	Enter a file name for the new package. Click the browse button to navigate to the folder where the new package will reside.
Description	Enter a description for the new package.

3.2.3 Database

Topics

Name	Description
Add Fields (↗ see page 739)	Use this dialog box to create a persistent field component for a dataset. Each time you open the dataset, the product verifies that each non-calculated persistent field exists or can be created from data in the database, and then creates persistent components for the fields you specified. If it cannot, it raises an exception warning you that the field is not valid, and does not open the dataset. After you have created a field component for the dataset, the product no longer creates dynamic field components for every column in the underlying database. **Tip:** To select multiple fields, press CTRL... more (↗ see page 739)
Assign Local Data (↗ see page 739)	Use this dialog box to copy the current set of records from another dataset to the selected client dataset. This is useful when populating client datasets for use as lookup tables, or when testing client datasets at design-time. Select the dataset you want to copy from the list of datasets available to the current form, then click **OK**. To clear the records in a client dataset at design-time, set the dataset's Active property to true, then right-click the client dataset and choose Clear Data.
Columns Collection Editor (↗ see page 740)	Use this dialog to to create and remove columns in a dataset DataTable object and to configure the properties for dataset columns. If you set your data adapter Active property to **True**, the names of the dataset columns, as returned from the database, appear in the members list.
Constraints Collection Editor (↗ see page 740)	Use this dialog to add, configure, and remove constraints to or from a dataset's DataTable columns. You can manage unique constraints and foreign key constraints in this editor.
Relations Collection Editor (↗ see page 740)	Use this dialog to create, edit, and delete relationships between the tables in the current dataset.
Tables Collection Editor (↗ see page 741)	Use this dialog to add and remove DataTable objects to or from a dataset and to set the properties for each DataTable object. The members list automatically displays one member for each DataTable in the corresponding dataset only if the Active property is set to **True** for the data adapter.
CommandText Editor (↗ see page 741)	Use this dialog box to construct the command text for the CommandText property of the BdpCommand.
Command Text Editor (↗ see page 742)	Use this dialog box to construct the command text (SQL statement) for dataset components that have a CommandText property. The dialog buttons add SELECT and FROM clauses; other clauses (WHERE, GROUP BY, HAVING, ORDER BY, and so on) must be added manually to the **SQL** edit control.
Configure Data Adapter (↗ see page 742)	Use this dialog box to construct the command text for the DataAdapter property of the BdpDataAdapter.
Connection Editor (↗ see page 743)	Use this dialog box to select a connection configuration or to edit the named connections that are stored in the BdpConnection.xml file. This editor lets you add, delete, and test your connection.

Connection Editor (see page 743)	Use this dialog box to select a connection configuration for a TSQLConnection component or to edit the named connections that are stored in the dbxconnections.ini file. Any changes you make in the dialog are written to the dbxconnections.ini file when you click **OK**. The selected connection is also assigned as the value of the SQL Connection component's ConnectionName property.
Connection String Editor (ADO) (see page 744)	Use this dialog box to specify the connection string used to connect an ADO data component to an ADO data store. You can type the connection string, build it using an ADO-supplied dialog box, or place the string in a file.
Data Adapter Dataset (see page 744)	Use this tab to select the dataset to associate with the Data Adapter.
DataAdapter Preview (see page 744)	Use this dialog box to preview the result set that is returned from the current SQL Select statement. By reviewing the result set, you can tune your SQL statements to provide a more accurate result set, which you can move into or out of the DataSet. This dialog is accessed from the **CommandText Editor** and the **Configure Data Adapter** dialog.
Database Editor (see page 745)	Use this dialog box to set up the connection to a database.
Database Form Wizard (see page 745)	**Other ▶ New ▶ Other ▶ Delphi Projects ▶ Business ▶ Database From Wizard** Use this wizard to create a form that displays data from a local or remote database. The wizard will connect the form to a TTable or TQuery component, write SQL statements for TQuery components, define the form tab order, and connect TDataSource components to TTable/TQuery components. Follow the instructions on each wizard page. Click **Next** to continue to the next page and then click **Finish** to generate the form based on the information you have provided. **Tip:** The Database Form Wizard is also available from the **Database ▶ Form Wizard** menu.
Dataset Properties (see page 746)	Use this dialog to examine the tables, columns, and other properties of a dataset. The left pane of the dialog displays the dataset and its tables. Select an object in the left pane to display its properties in a read-only grid in the right pane.
Driver Settings (see page 746)	Use this dialog box to see what files are associated with each dbExpress driver name. These associations are stored in the drivers.ini file.
Field Link Designer (see page 746)	Use this dialog box to establish a master-detail relationship between two tables.
Fields Editor (see page 746)	Use this dialog box to add new persistent fields to a dataset and create data fields, calculated fields, and lookup fields. The dialog box is displayed when you right-click any of several dataset components, such as TADODataSet or TSQLDataSet, and choose Fields Editor. Use the **Fields Editor** at design time to create persistent lists of the field components used by the datasets in your application. Persistent fields component lists are stored in your application, and do not change even if the structure of a database underlying a dataset is changed. All fields in a dataset are either persistent or dynamic.... more (see page 746)
Foreign Key Constraint (see page 747)	Use this dialog box to define a foreign key constraint between tables in a dataset. The table you selected when navigating to this dialog is automatically set to the foreign key table. You could, therefore, attempt to set the master table in a master-detail relationship to be the detail table. This could, at the very least return unpredictable results. If the actual detail, or foreign key, table's foreign key column contains duplicates, you may generate a design-time error.
Generate Dataset (see page 748)	Use this dialog box to sort records in a dataset, locate records quickly, limit records that are visible, and establish master/detail relationships.
New Connection (see page 748)	Use this dialog box to create a new named database connection configuration.
New Field (see page 748)	Use this dialog box to create persistent fields in a dataset. You can add persistent fields or replace existing persistent fields.
Relation (see page 749)	Use this dialog box to define a relationship between tables in a dataset.
Rename Connection (see page 749)	Enter a new name for the currently selected named connection in the **Connection Builder**. Renaming a connection updates the dbxconnections.ini file to change the name for the selected connection configuration. The old name becomes unavailable and can no longer be used as the ConnectionName property of a TSQLConnection component.
SQL Monitor (see page 749)	**Database ▶ SQL Monitor** Use this dialog box to see the actual statement calls made through SQL Links to a remote server or through the ODBC socket to an ODBC data source. You can monitor different types of activity by choosing **Options ▶ Trace Options** to display the **Trace Options** dialog box.
Sort Fields Editor (see page 750)	Use this dialog box to specify the fields used to sort records in a SQL dataset, which has the CommandType property set to ctTable.

Stored Procedures Dialog (↗ see page 751)	Use this dialog box to select a stored procedure for a BdpCommand. You can specify values for Input or InputOutput parameters and execute the selected stored procedure.
TableMappings Collection Editor (↗ see page 751)	Use this dialog box to map columns between your data source and the in-memory dataset.
Unique Constraint (↗ see page 752)	Use this dialog box to select the dataset columns you want included in your constraint, which means these columns will be forced to contain only unique values. You can also set one or more columns as your primary key.
IBDatabase Editor dialog box (↗ see page 752)	The Database Editor dialog box sets up the properties of a database that specify the connection that should be made to a database. This dialog box allows you to specify the type of database, the connection parameters, the user name, SQL role, and password, and whether or not a login prompt is required. These properties of the database component, as well as others, can also be specified using the Object Inspector. To display the Database Editor dialog box, double click on an IBDatabase component.
IBTransaction Editor dialog box (↗ see page 753)	The Transaction Editor dialog box allows you to set up transaction parameters. This dialog box gives you four default transaction settings, which you can then customize if you wish. Once you modify the default transaction, the radio button is unset. For a complete list of all the InterBase transaction parameters, refer to 'Working with Transactions' in the InterBase API Guide. These properties of the transaction component, as well as others, can also be specified using the Object inspector. To display the Transaction Editor dialog box, double click on an IBTransaction component. The following four choices are displayed:
IBUpdateSQL and IBDataSet Editor dialog box (↗ see page 753)	Use the editor to create SQL statements for updating a dataset. The TIBUpdateSQL object must be associated with a TIBQuery object by setting the TIBQuery property UpdateObject to the name of the TIBUpdateSQL object used to contain the SQL statements. A datasource and database name must be selected for the TIBQuery object. In addition, the SQL property must include an SQL statement defining a table. To open the SQL editor: 1. Select the TIBUpdateSQL or TIBDataSet object in the form. 2. Right-click and choose Update SQL editor or DataSet editor. The Update SQL editor has two pages, the Options page and the... more (↗ see page 753)

3.2.3.1 Add Fields

Use this dialog box to create a persistent field component for a dataset.

Each time you open the dataset, the product verifies that each non-calculated persistent field exists or can be created from data in the database, and then creates persistent components for the fields you specified. If it cannot, it raises an exception warning you that the field is not valid, and does not open the dataset.

After you have created a field component for the dataset, the product no longer creates dynamic field components for every column in the underlying database.

Tip: To select multiple fields, press CTRL

and click the fields. To select a range of fields, press `SHIFT` and click the first and last fields in the range.

3.2.3.2 Assign Local Data

Use this dialog box to copy the current set of records from another dataset to the selected client dataset. This is useful when populating client datasets for use as lookup tables, or when testing client datasets at design-time. Select the dataset you want to copy from the list of datasets available to the current form, then click **OK**. To clear the records in a client dataset at design-time, set the dataset's Active property to true, then right-click the client dataset and choose Clear Data.

3.2.3.3 Columns Collection Editor

Use this dialog to to create and remove columns in a dataset DataTable object and to configure the properties for dataset columns. If you set your data adapter Active property to **True**, the names of the dataset columns, as returned from the database, appear in the members list.

Item	Description
Add	Adds a column to the dataset DataTable object.
Remove	Removes the selected column from the dataset DataTable object.
Properties	Contains a list of database properties that you can apply to the dataset columns. These may change based on the database type.
Properties | ColumnMapping	If you are mapping your dataset columns to an XML file, set this property to determine whether the column value should be treated as an element, an attribute, simple content, or if it should be hidden.

See Also

Using Standard Datasets

3.2.3.4 Constraints Collection Editor

Use this dialog to add, configure, and remove constraints to or from a dataset's DataTable columns. You can manage unique constraints and foreign key constraints in this editor.

Item	Description
Add	Adds either a unique constraint or a foreign key constraint to the DataTable column.
Edit	This button appears after you have added a constraint. It allows you to change the columns specified in the constraint.
Remove	Removes the selected constraint from a DataTable column.
Properties	The properties list is read-only, because all properties displayed here are derived from existing dataset and column definitions. You can, however, expand selections in this list to view the specific properties for a given column, and for the constraint itself.

See Also

Using Standard Datasets

3.2.3.5 Relations Collection Editor

Use this dialog to create, edit, and delete relationships between the tables in the current dataset.

Item	Description
Add	Adds a new relation to the relation collection. Displays the **Relation** dialog, allowing you to specify a relationship name, keys, and rules for tables in your dataset.
Edit	Displays the **Relation** dialog, allowing you to change an existing table relation.

Remove	Removes the selected relation from the member list.
Properties	Provides a read-only list of properties that are derived from the dataset and related columns that comprise the relation.

See Also

Using Standard Datasets

3.2.3.6 Tables Collection Editor

Use this dialog to add and remove DataTable objects to or from a dataset and to set the properties for each DataTable object. The members list automatically displays one member for each DataTable in the corresponding dataset only if the Active property is set to **True** for the data adapter.

Item	Description
Add	Adds a DataTable object to the dataset.
Remove	Removes the selected DataTable object from the dataset.

See Also

Using Standard Datasets

3.2.3.7 CommandText Editor

Use this dialog box to construct the command text for the CommandText property of the BdpCommand.

Item	Description
Connection	Specifies a connection from a list of live BdpConnection objects. You must select this item first to populate the other text boxes with usable data.
Select	Specifies that a SQL Select statement is to be generated by the BdpCommandBuilder. This statement must exist for the BdpCommandBuilder to be able to construct other statements, such as Update, Insert, or Delete statements. You can either supply your own Select statement in the BdpCommand object, or use this dialog to create one.
Update	Specifies that the BdpCommandBuilder is to generate an Update statement based on the Select statement.
Insert	Specifies that the BdpCommandBuilder is to generate an Insert statement based on the Select statement.
Delete	Specifies that the BdpCommandBuilder is to generate a Delete statement based on the Select statement.
Generate SQL	Generates SQL statements for the selected check box items.
Optimize	Specifies that, when possible, the BdpCommandBuilder is to generate optimized SQL statements for the selected check box items.
Tables	Displays a list of tables from the current database, represented by the current BdpConnection. To display a list of columns, select a table from the list. Select a table name to include it in the SQL statement that is currently visible in the SQL text box.
Columns	Displays a list of columns from a selected table. Select one or more column names to include them in the SQL statement that is currently visible in the SQL text box.

3.2.3.8 Command Text Editor

Use this dialog box to construct the command text (SQL statement) for dataset components that have a CommandText property. The dialog buttons add SELECT and FROM clauses; other clauses (WHERE, GROUP BY, HAVING, ORDER BY, and so on) must be added manually to the **SQL** edit control.

Item	Description
Tables	Displays the names of tables available in the current database. Select a table and click the **Add Table to SQL** button to add it to the command (SQL statement) displayed in the **SQL** edit control.
Add Table to SQL	Adds a SELECT clause for the table name selected in the **Tables** list to **SQL** edit box.
Fields	Displays the names of the columns available in the table currently highlighted in the **Tables** list. Select one or more columns and click the **Add Field to SQL** button to add it to the command (SQL statement) displayed in the **SQL** edit control. To select multiple columns, press `CTRL` and click the columns. To select a range of columns, press `SHIFT` and click the first and last columns in the range.
Add Field to SQL	Adds the columns selected in the **Fields** list to the SELECT clause.
SQL	Displays the command (SQL statement) for the CommandText property of the dataset or command component. This statement can be edited manually.

3.2.3.9 Configure Data Adapter

Use this dialog box to construct the command text for the DataAdapter property of the BdpDataAdapter.

Item	Description
Connection	Specifies a connection from a list of live BdpConnections. You must select this item first to populate the other text boxes with usable data.
Select	Specifies that a SQL Select statement is to be generated by the BdpCommandBuilder. This statement must exist for the BdpCommandBuilder to be able to construct other statements, such as Update, Insert, or Delete statements. You can either supply your own Select statement in the BdpCommand object, or use this dialog to create one.
Update	Specifies that the BdpCommandBuilder is to generate an Update statement based on the Select statement.
Insert	Specifies that the BdpCommandBuilder is to generate an Insert statement based on the Select statement.
Delete	Specifies that the BdpCommandBuilder is to generate a Delete statement based on the Select statement.
Generate SQL	Generates SQL statements for the selected check box items.
Optimize	Specifies that, when possible, the BdpCommandBuilder is to generate optimized SQL statements for the selected check box items.
Tables	Displays a list of tables from the current database, represented by the current BdpConnection. To display a list of columns, select a table from the list. Select a table name to include it in the SQL statement that is currently visible in the SQL text box.
Columns	Displays a list of columns from a selected table. Select one or more column names to include them in the SQL statement that is currently visible in the SQL text box.

3.2.3.10 Connection Editor

Use this dialog box to select a connection configuration or to edit the named connections that are stored in the BdpConnection.xml file. This editor lets you add, delete, and test your connection.

Item	Description
Connections	Shows all the named connection configurations for the currently selected driver. When you select a connection from this list box, the Connection Settings table displays the connection parameters for that named connection.
Connection Settings	Lists the connection parameters for the currently selected connection name. The Name column lists the names of connection parameters that are applicable to the driver associated with the current connection. The Value column changes the value of any connection parameter.
Add	Lets you add a new named connection. Click the **Add** button to display the **Add New Connection** dialog box to specify the name for the connection configuration and indicate what drivers to use.
Remove	Deletes the connection currently selected in the **Connections** list box. Deleting a named connection removes it from the dbxconnections.ini file so that it is no longer available for use by any application.
Test	Attempts to establish a connection to the database server using the currently specified connection settings. Click this button to check that the currently defined connection configuration can be used.

3.2.3.11 Connection Editor

Use this dialog box to select a connection configuration for a TSQLConnection component or to edit the named connections that are stored in the dbxconnections.ini file. Any changes you make in the dialog are written to the dbxconnections.ini file when you click **OK**. The selected connection is also assigned as the value of the SQL Connection component's ConnectionName property.

Item	Description
Add Connection	Displays the **New Connection** dialog, allowing you to specify the name for the connection configuration and indicate what driver it uses. The new name connection configuration is added to the dbxconnections.ini file.
Delete Connection	Deletes the connection currently selected in the **Connections** list box. Deleting a named connection removes it from the dbxconnections.ini file so that it is no longer available for use by any application.
Rename Connection	Displays the **Rename connection** dialog, where you can specify a new name for the connection currently selected in the **Connections** list box. Renaming a connection configuration renames it in the dbxconnections.ini file so that the old name is no longer available for use by any application.
Test Connection	Attempts to establish a connection to the database server using the currently specified connection settings. Use this button to check that the currently defined connection configuration can be used.
View Driver Settings	Displays the **Driver Settings** dialog, where you can obtain information about the currently installed drivers.
Driver Name	Lists all of the drivers for which there is a connection defined. When you select a driver, the **Connections** list box displays only the names of connection configurations for the selected driver.
Connection Settings	Lists the connection parameters for the currently selected connection name. The **Key** column lists the names of all connection parameters that are applicable to the driver associated with the current connection. You can select entries in the **Value** column to change the value of any connection parameter. Do not change the driver name for a connection. If you change the driver name, the listed connection parameters will not be appropriate for the new driver you specify.

Connection Name	Lists all of the named connection configurations for the currently selected driver. You can select a connection name and edit the connection settings to make changes to the configuration stored in the dbxconnections.ini file, or simply select a connection and click **OK** to assign that connection configuration as the value of the ConnectionName.

3.2.3.12 Connection String Editor (ADO)

Use this dialog box to specify the connection string used to connect an ADO data component to an ADO data store. You can type the connection string, build it using an ADO-supplied dialog box, or place the string in a file.

Item	Description
Use Data Link File	Select or enter the name of a data link file, or click the **Browse** button to locate the file.
Use Connection String	Enter the connection string with the connection information in the edit box. Alternatively, click the **Build** button to display the **Data Link Properties** dialog to guide you through setting up and testing the connection.

Tip: ADO supports the following four arguments for connection strings: provider, file name, remote provider, and remote server. Any other arguments are passed on to the provider. Such parameters might include user ID, login password, the name of a default database, persistent security information, ODBC data source names, connection time-out values, and locale identifiers. These parameters and their values are specific to particular providers, servers, and ODBC drivers and not to either ADO or Delphi. For specific information on them, consult the documentation for the provider, server, or ODBC driver.

3.2.3.13 Data Adapter Dataset

Use this tab to select the dataset to associate with the Data Adapter.

Item	Description
New Data Set	Specifies a new data set if you do not have an existing one.
Existing Data Set	Specifies an existing dataset associated with the Data Adapter.
None	Does not specify any dataset.

3.2.3.14 DataAdapter Preview

Use this dialog box to preview the result set that is returned from the current SQL Select statement. By reviewing the result set, you can tune your SQL statements to provide a more accurate result set, which you can move into or out of the DataSet. This dialog is accessed from the **CommandText Editor** and the **Configure Data Adapter** dialog.

Item	Description
Limit Rows	Specifies a rowlimit option on the current SQL statement.
Rows to fetch	Sets the actual rowlimit maximum.
Refresh	Refreshes the result set displayed in the list, by executing the SQL statement again.

3.2.3.15 Database Editor

Use this dialog box to set up the connection to a database.

Item	Description
Name	Specifies the name of the database. This name refers to the database component from within the code of your application.
Alias Name	Specifies the BDE alias for the database. Choose a database aliases from the drop-down list. This list contains all aliases currently registered with the BDE. If you do not want to connect to a database that is registered as a BDE alias, you can set the Driver property instead. If you set the Alias property, the Driver property is cleared, as the driver type is implicit in the BDE alias.
Driver Name	Specifies the type of database represented by the database component. Choose a driver type such as STANDARD, ORACLE, SYBASE, or INTERBASE from the drop-down list. If the database server has an alias registered with the BDE, you can set the Alias instead. Setting the **Driver** automatically clears the Alias property, to avoid potential conflicts with the driver type implicit in the database alias.
Parameter overrides	Specifies the values of all login parameters when connecting to the database. The specific parameters depend on the type of database. To obtain a list of all parameters, as well as their default values, click the **Defaults** button. You can then modify the default values to the values you want to use.
Defaults	Sets the Parameter overrides to the default values for the driver type.
Clear	Removes all parameter overrides.
Login Prompt	Causes a login dialog to appear automatically when the user connects to the database. Uncheck the **Login Prompt** control to prevent the automatic login dialog. Most database servers (except for the file-based STANDARD types) require the user to supply a password when connecting to the database. For such servers, if the automatic login prompt is omitted, the application must supply the user name and password in some other manner. These can be supplied either by providing hard-coded parameter overrides, or by supplying an OnLogin event handler that sets the values for these parameters.
Keep inactive connection	Indicates that the application should remain connected to the database even if no datasets are currently open. For connections to remote database servers, or for applications that frequently open and close datasets, checking **Keep inactive connection** reduces network traffic, speeds up applications, and avoids logging in to the server each time the connection is reestablished. Uncheck **Keep inactive connection** to cause the database connection to be dropped when there are no open datasets. Dropping a connection releases system resources allocated to the connection, but if a dataset is later opened that uses the database, the connection must be reestablished and initialized.

3.2.3.16 Database Form Wizard

Other ▶New▶Other▶Delphi Projects▶Business▶Database From Wizard

Use this wizard to create a form that displays data from a local or remote database. The wizard will connect the form to a TTable or TQuery component, write SQL statements for TQuery components, define the form tab order, and connect TDataSource components to TTable/TQuery components.

Follow the instructions on each wizard page. Click **Next** to continue to the next page and then click **Finish** to generate the form based on the information you have provided.

Tip: The Database Form Wizard

is also available from the **Database▶Form Wizard** menu.

3.2.3.17 Dataset Properties

Use this dialog to examine the tables, columns, and other properties of a dataset.

The left pane of the dialog displays the dataset and its tables. Select an object in the left pane to display its properties in a read-only grid in the right pane.

3.2.3.18 Driver Settings

Use this dialog box to see what files are associated with each dbExpress driver name. These associations are stored in the drivers.ini file.

Item	Description
Driver Name	Lists the names of all drivers that appear in the drivers.ini file. These drivers represent the possible values of the DriverName parameter in the Connection Editor, which determines the DriverName property of a TSQLConnection component.
Library Name	Displays the dbExpress driver file for the driver named in the Driver Name column. It determines the LibraryName property of a TSQLConnection component that is added automatically when you set the DriverName property.
Vendor Library	Displays the client-side DLL for the database server associated with the specified driver. This DLL is supplied by the database vendor. The value of Vendor Library determines the VendorLib property of a TSQLConnection component that is added automatically when you set the DriverName property.

3.2.3.19 Field Link Designer

Use this dialog box to establish a master-detail relationship between two tables.

Item	Description
Available Indexes	(Not displayed for tables on a database server.) Shows the currently selected index used to join the tables. Unless you specify a different index name in the table’s IndexName property, the default index used for the link is the primary index for the table. Other available indexes defined on the table can be selected from the drop-down list.
Detail Fields	Lists the details fields in the table.
Add	Adds the selected detail and master fields to the **Joined Fields** box.
Master Fields	Lists the master fields in the table.
Joined Fields	Displays the selected master and detail fields.
Delete	Removes the selected line from the **Joined Fields** box.
Clear	Removes all of the lines in the **Joined Fields** box..

3.2.3.20 Fields Editor

Use this dialog box to add new persistent fields to a dataset and create data fields, calculated fields, and lookup fields. The dialog box is displayed when you right-click any of several dataset components, such as TADODataSet or TSQLDataSet, and choose Fields Editor.

Use the **Fields Editor** at design time to create persistent lists of the field components used by the datasets in your application. Persistent fields component lists are stored in your application, and do not change even if the structure of a database underlying a dataset is changed. All fields in a dataset are either persistent or dynamic.

Context Menu

Right-click the **Fields Editor** to access the following commands.

Item	Description
Add fields	Displays the **Add Fields** dialog box which enables you to add persistent field components for a dataset.
New field	Displays the **New Field** dialog box which enables you to create new persistent fields as additions to or replacements of the other persistent fields in a dataset. The types of new persistent fields that may be created are data fields, calculated fields, and lookup fields.
Add all Fields	Creates persistent fields for every field in the underlying dataset.
Cut	Removes the selected fields from the **Fields Editor** and places them on the clipboard.
Paste	Pastes the clipboard contents into the **Fields Editor**.
Delete	Deletes selected fields without copying them to the clipboard.
Select All	Selects all of the fields in the **Fields Editor**.

3.2.3.21 Foreign Key Constraint

Use this dialog box to define a foreign key constraint between tables in a dataset. The table you selected when navigating to this dialog is automatically set to the foreign key table. You could, therefore, attempt to set the master table in a master-detail relationship to be the detail table. This could, at the very least return unpredictable results. If the actual detail, or foreign key, table's foreign key column contains duplicates, you may generate a design-time error.

Item	Description
Name	Specify the name of the relation.
Parent table	Specify the parent table in the relation.
Child table	Specifies the child table in the relation. This value is determined by the table you selected to get to this dialog. It is read-only.
Key Columns	Select one or more columns to act as the primary key in the parent table.
Foreign Key Columns	Select one or more columns to act as the foreign key in the child table. Typically, these must correspond to the columns you chose as your primary key, if not in name, then by data type and value.
Update rule	Select the Update rule to use when updating records. Applies to master-detail relationships and how detail records are updated when a master record is updated.
Delete rule	Select the Delete rule to use when deleting records. Applies to master-detail relationships and how detail records are deleted when a master record is deleted.
Accept/Reject rule	Select the Accept/Reject rule to use when an insert occurs in a master-detail relationship.

See Also

Using Standard Datasets

3.2.3.22 Generate Dataset

Use this dialog box to sort records in a dataset, locate records quickly, limit records that are visible, and establish master/detail relationships.

Item	Description
Existing	Specifies an existing dataset previously created.
New	Specifies a new dataset.
Choose which table(s) to add to the dataset	Selects a table or multiple tables to populate the dataset.

3.2.3.23 New Connection

Use this dialog box to create a new named database connection configuration.

Item	Description
Driver Name	Shows all of the drivers listed in the drivers.ini file. Select the driver the new connection will use.
Connection Name	Enter a name for the new named connection. This name should be unique: that is, it should not be the same as any other connection name listed in the dbxconnections.ini file.

3.2.3.24 New Field

Use this dialog box to create persistent fields in a dataset. You can add persistent fields or replace existing persistent fields.

Item	Description
Name	Enter the component's field name.
Component	Displays a name you enter in the **Name** field, prefixed with the component name. The product discards anything you enter directly in the **Component** edit box.
Type	Select the field component's data type. You must supply a data type for any new field component you create. For example, to display floating point currency values in a field, select **Currency** from the drop-down list.
Size	Specify the maximum number of characters that can be displayed or entered in a string-based field or the size of Bytes and VarBytes fields. For all other data types, **Size** is meaningless.
Data	Replaces an existing field (for example to change its data type) and is based on columns in the table or a query underlying a dataset.
Calculated	Displays values calculated at runtime by a dataset's OnCalcFields event handler.
Lookup	Retrieve values from a specified dataset at runtime based on search criteria you specify (not supported for unidirectional datasets).
InternalCalc	Retrieves values calculated at runtime by a client dataset and stored with its data (instead of being dynamically calculated in an OnCalcFields event handler). InternalCalc is only available if you are working with a client dataset. Values calculated for an InternalCalc field are stored and retrieved as part of the client dataset's data.
Aggregate	Retrieves a value summarizing the data in a set of records from a client dataset.

Key Fields	Lists the fields in the current dataset for which you can match values. To match more than one field, enter field names directly instead of choosing from the drop-down list. Separate multiple field names with semicolons. If you are using more than one field, you must use persistent field components.
Dataset	Lists the datasets in which to look up field values. The lookup dataset must be different from the dataset for the field component itself, or a circular reference exception is raised at runtime.
Lookup Keys	Used only to create lookup fields. If you choose **Lookup** as the field type, the **Key Fields** and **Dataset** edit boxes in the Lookup definition group box are enabled. The **Lookup Keys** and **Result Field** boxes are only enabled if you are connected to a second dataset.
Result Field	Lists the fields in the lookup dataset that can be returned as the value of the lookup field you are creating.

3.2.3.25 Relation

Use this dialog box to define a relationship between tables in a dataset.

Item	Description
Name	Specify the name of the relation.
Parent table	Specify the parent table in the relation.
Child table	Specify the child table in the relation.
Key Columns	Select one or more columns to act as the primary key in the parent table.
Foreign Key Columns	Select one or more columns to act as the foreign key in the child table. Typically, these must correspond to the columns you chose as your primary key, if not in name, then by data type and value.
Update rule	Select the Update rule to use when updating records. Applies to master-detail relationships and how detail records are updated when a master record is updated.
Delete rule	Select the Delete rule to use when deleting records. Applies to master-detail relationships and how detail records are deleted when a master record is deleted.
Accept/Reject rule	Select the Accept/Reject rule to use when an insert occurs in a master-detail relationship.

See Also

Using Standard Datasets

3.2.3.26 Rename Connection

Enter a new name for the currently selected named connection in the **Connection Builder**. Renaming a connection updates the dbxconnections.ini file to change the name for the selected connection configuration. The old name becomes unavailable and can no longer be used as the ConnectionName property of a TSQLConnection component.

3.2.3.27 SQL Monitor

Database ▶ SQL Monitor

Use this dialog box to see the actual statement calls made through SQL Links to a remote server or through the ODBC socket to an ODBC data source.

You can monitor different types of activity by choosing **Options ▶ Trace Options** to display the **Trace Options** dialog box.

Trace Options - Category Page

Use this page to select the trace categories to be monitored.

Item	Description
Prepared Query Statements	Prepared statements to be sent to the server.
Executed Query Statements	Statements to be executed by the server. Note that a single statement may be prepared once and executed several times with different parameter bindings.
Input Parameters	Parameter data sent to servers when doing INSERTs or UPDATEs.
Fetched Data	Data retrieved from servers.
Statement Operations	Each operation performed such as ALLOCATE, PREPARE, EXECUTE, and FETCH
Connect / Disconnect	Operations associated with connecting and disconnecting to databases, including allocation of connection handles, freeing connection handles, if required by server.
Transactions	Transaction operations such as BEGIN, COMMIT, and ROLLBACK (ABORT).
Blob I/O	Operations on Blob datatypes, including GET BLOB HANDLE, STORE BLOB, and so on.
Miscellaneous	Operations not covered by other categories.
Vendor Errors	Error messages returned by the server. The error message may include an error code, depending on the server.
Vendor Calls	Actual API function calls to the server. For example, ORLON for Oracle, ISC_ATTACH for InterBase.

Trace Options - Buffer Page

Use this page to manage the trace buffer maintained in memory by the SQL Monitor.

Item	Description
Buffer Size	Indicates the size of the buffer to be used for trace information.
Circular	Writes trace information in a circular memory buffer, so that once the limit is reached, additional traces replace the first traces.
Page to disk	Writes trace information to a disk file when the memory buffer becomes full.
Filename	Indicates the name of the trace file to be used if **Page to disk** is selected.

3.2.3.28 Sort Fields Editor

Use this dialog box to specify the fields used to sort records in a SQL dataset, which has the CommandType property set to ctTable.

Item	Description
Available Fields	Lists all of the fields in the database table. Select fields in this list and use the arrow buttons to move them to the list of fields on which to sort.
Order by Fields	Displays the fields you have selected. The SQL dataset sorts its records by the first field. Within groups defined by the first field, they are sorted by the second field, and so on.
Arrow buttons	Use the arrow buttons to move fields between the two lists.

3.2.3.29 Stored Procedures Dialog

Use this dialog box to select a stored procedure for a BdpCommand. You can specify values for Input or InputOutput parameters and execute the selected stored procedure.

Item	Description
Stored Procedures	Shows all the stored procedures available for the BdpConnection associated with the BdpCommand. When you select a stored procedure from this drop-down list box, the Parameters list box displays the parameters for that stored procedure.
Stored Procedure Has One Or More Resultset	If the stored procedure returns one or more cursors, check this check box to display the resultset in the dialog box when the stored procedure is executed.
Parameters	Shows all the parameters for the currently selected stored procedure. When you select a parameter from this list box, the associated metadata appear and can be edited in the pane on the right.
Execute	Attempts to execute the selected stored procedure using the currently specified parameter settings. The results of the stored procedure execution (Output parameters, InputOutput parameters, return values, cursor(s) returned) are all populated into a DataGrid in the bottom of the dialog box.

3.2.3.30 TableMappings Collection Editor

Use this dialog box to map columns between your data source and the in-memory dataset.

Item	Description
Use a dataset to suggest table and column names	By checking this option, you are presented with a list of available typed datasets, if any, which will provide existing table and column names. Returns only a list of names from the schema of the given typed dataset. Does not apply to standard datasets.
Dataset	Displays the names of datasets that you can use for a model.
Source table	Displays the name of the data source table. Because a data adapter can reference more than one table, you can select from multiple tables in the data source, if available.
Dataset table	Displays the name of the dataset table. Because you can create datasets that contain multiple tables, you might see multiple tables listed here.
Source columns	Displays the names of all columns in the data source table.
Dataset columns	Displays the names of columns in the dataset to write the data source table columns to. When performing updates to the data source, these are the columns that will be read from. You can change the names to map one column to another non-corresponding column, or you can map to corresponding columns between the source table and the dataset.
Delete	Deletes the active column names in both source and dataset column lists. Deleted columns will not appear in the dataset. Useful if your query returns more columns from the data source than you need to use in the dataset.
Reset	Resets the list of source table column names and dataset column names to their original values.
Enter key	You can add a row by pressing Enter while the cursor is in the last row of the dataset column. This allows you to create new columns which might exist already but don't display at designtime, for some reason. For example, if you have derived or computed fields that you want to keep track of, you could add a new column to account for that data. The order of columns does not have any impact on the data.

3.2.3.31 Unique Constraint

Use this dialog box to select the dataset columns you want included in your constraint, which means these columns will be forced to contain only unique values. You can also set one or more columns as your primary key.

Item	Description
Name	Specifies the name of the constraint.
Columns	Check each column you want included in the constraint. If you indicate a nonunique column as unique, you may generate a runtime errror.
Primary key	Selecting this option sets the checked columns in your constraint as the primary key for the current DataTable. If you set a combination of columns, some of which contain duplicates or null values, you may receive a design-time error and a runtime error. However, if you receive a design-time error, the constraint may still be defined as you indicated, even if it is incorrect.

3.2.3.32 IBDatabase Editor dialog box

The Database Editor dialog box sets up the properties of a database that specify the connection that should be made to a database. This dialog box allows you to specify the type of database, the connection parameters, the user name, SQL role, and password, and whether or not a login prompt is required.

These properties of the database component, as well as others, can also be specified using the Object Inspector.

To display the Database Editor dialog box, double click on an IBDatabase component.

Dialog box options

Connection

Option	Meaning
Local	Indicates that the database is on the local server. Enables the Browse button, allowing you to search for the database with a Open File dialog.
Remote	Indicates that the database is on a remote server. Activates the Protocol and Server fields
Protocol	Sets the protocol for attaching to the remote server. The protocol can be TCP/IP, Named Pipe, or SPX.
Server	The name of the remote server.
Database	The name of the database.

Database Parameters

Option	Meaning
User Name	The name of the database user.
Password	The password for the database user.
SQLRole	The SQLRole name used to connect to the database.
Character Set	The character set used to connect to the database.
Login Prompt	Indicates whether a login prompt is required to access the database.

Settings	Displays the current parameters and allows you to add other parameters.
	For example:
	`user_name=sysdba`
	`password=masterkey`
	`sql_role_name=finance`
	`lc_ctype=WIN1252`
	For more information on database parameters, see the InterBase API Guide.

3.2.3.33 IBTransaction Editor dialog box

The Transaction Editor dialog box allows you to set up transaction parameters. This dialog box gives you four default transaction settings, which you can then customize if you wish. Once you modify the default transaction, the radio button is unset.

For a complete list of all the InterBase transaction parameters, refer to 'Working with Transactions' in the InterBase API Guide.

These properties of the transaction component, as well as others, can also be specified using the Object inspector.

To display the Transaction Editor dialog box, double click on an IBTransaction component. The following four choices are displayed:

Snapshot

By default, Snapshot is set to concurrency and no wait, which means that the transaction is aware of other transactions, and does not wait for locks to be released, returning an error instead.

Read Committed

By default, Read Committed is set to read_committed, rec_version, and no wait, which means that the transaction reads changes made by concurrent transactions, can read the most recently committed version of a transaction, and does not wait for locks to be released, returning an error instead.

Read-Only Table Stability

By default, Read-Only Table Stability is set to read and consistency, which means that the transaction can read a specified table and locks out other transactions.

Read-Write Table Stability

By default, Read-Write Table Stability is set to write and consistency, which means that the transaction can read and write to a specified table and locks out other transactions.

For a complete list of all the InterBase transaction parameters, refer to 'Working with Transactions' in the InterBase API Guide.

3.2.3.34 IBUpdateSQL and IBDataSet Editor dialog box

Use the editor to create SQL statements for updating a dataset.

The TIBUpdateSQL object must be associated with a TIBQuery object by setting the TIBQuery property UpdateObject to the name of the TIBUpdateSQL object used to contain the SQL statements. A datasource and database name must be selected for the TIBQuery object. In addition, the SQL property must include an SQL statement defining a table.

To open the SQL editor:

1. Select the TIBUpdateSQL or TIBDataSet object in the form.
2. Right-click and choose Update SQL editor or DataSet editor.

The Update SQL editor has two pages, the Options page and the SQL page.

The Options page

The Options page is visible when you first invoke the editor.

Table Name	Use the Table Name combo box to select the table to update. When you specify a table name, the Key Fields and Update Fields list boxes are populated with available columns.
Key Fields	The Key Fields list box is used to specify the columns to use as keys during the update. Generally the columns you specify here should correspond to an existing index.
Update Fields	The Update Fields list box indicates which columns should be updated. When you first specify a table, all columns in the Update Fields list box are selected for inclusion. You can multi-select fields as desired.
Get Table Fields	Read the table fields for the table name entered and list the fields.
Dataset Defaults	Use this button to restore the default values of the associated dataset. This will cause all fields in the Key Fields list and the Update Fields list to be selected and the table name to be restored.
Select Primary Keys	Click the Primary Key button to select key fields based on the primary index for a table.
Generate SQL	After you specify a table, select key columns, and select update columns, click the Generate SQL button to generate the preliminary SQL statements to associate with the update component's ModifySQL, InsertSQL, DeleteSQL, and RefreshSQL properties.
Quote Identifiers	Check the box labeled Quote Field Names to specify that all field names in generated SQL be enclosed by quotation marks. This option is disabled at this time.

The SQL page

To view, modify, and refresh the generated SQL statements, select the SQL page. If you have generated SQL statements, then when you select this page, the statement for the ModifySQL property is already displayed in the SQL Text memo box. You can edit the statement in the box as desired.

Note: Note: Keep in mind that generated SQL statements are intended to be starting points for creating update statements. You may need to modify these statements to make them execute correctly. Test each of the statements directly yourself before accepting them.

Use the Statement Type radio buttons (Modify, Insert, Delete, or Refresh) to switch among generated SQL statements and edit them as desired.

To accept the statements and associate them with the update component's SQL properties, click OK.

3.2.4 Edit

Topics

Name	Description
Alignment (↗ see page 756)	**Edit ▶Align** Lines up selected components in relation to each other or to the form.
Creation Order (↗ see page 756)	**Edit ▶Creation Order** Specifies the order in which your application creates nonvisual components when you load the form at design time or runtime. The list box displays only those nonvisual components on the active form, their type, and their current creation order. The default creation order is determined by the order in which you placed the nonvisual components on the form.

Edit Tab Order (↗ see page 756)	**Edit ▶ Tab Order** Modifies the tab order of the components on the VCL Form, or within the selected component if that component contains other components. For example: a Form containing an edit control with the Tab property value of 0, and a button with the Tab property of 1, the edit control comes before the button when you cycle through the controls with the `Tab` key. The **Tab Order** dialog facilitates changing the values of the Tab property using the **Up** and **Down** arrows instead of explicitly having to specify values for the Tab property in the **Object Inspector**.... more (↗ see page 756)
Scale (↗ see page 757)	**Edit ▶ Scale** Proportionally resizes all the components on the current form. Scaling the form repositions the components as well as resizing them. Enter a percentage to which you want to resize the form's components. The scaling factor must be between 25 and 400. Percentages over 100 grow the form's components. Percentages under 100 shrink the form's components.
Size (↗ see page 757)	**Edit ▶ Size** Resizes multiple components to be exactly the same height or width.
Align to Grid (↗ see page 757)	**Edit ▶ Align to Grid** Aligns selected objects to the grid in the **Form Designer**. By default, components are aligned to a four-pixel grid used by the **Form Designer**. Press `ALT` to override this default when positioning components.
Bring to Front (↗ see page 757)	**Edit ▶ Bring to Front** Brings selected objects to the visual front of an active form. For example: if you have are working on a form in the **Form Designer** and you drop a TPanel over a portion of a TListbox, select Bring to Front to have the TListbox lie over the TPanel.
Copy (↗ see page 758)	**Edit ▶ Copy** Copies the current selection to the clipboard. Copy works with text when using the **Code Editor** and with components when using the **Form Designer**.
Cut (↗ see page 758)	**Edit ▶ Cut** Removes the current selection and stores it to the clipboard. Cut works with text when using the **Code Editor** and with components when using the **Form Designer**.
Delete (↗ see page 758)	**Edit ▶ Delete** Removes the selected object or group of objects. Delete works with text when using the **Code Editor** and with components when using the **Form Designer**. If you accidentally delete an object, use the **Edit ▶ Undo** command.
Flip Children (↗ see page 758)	**Edit ▶ Flip Children ▶ All** **Edit ▶ Flip Children ▶ Selected** Reverses Bi-Directional (BiDi) support. This is useful when you design a form for a language where the text reads right to left, such as Japanese or Arabic.
Lock Controls (↗ see page 758)	**Edit ▶ Lock Controls** Locks objects in place on an active form. You cannot move locked objects. To unlock, select Lock Controls again.
Paste (↗ see page 758)	**Edit ▶ Paste** Pastes contents of the clipboard previously caputured using Cut or Copy. Paste works with text when using the **Code Editor** and with components when using the **Form Designer**.
Select All (↗ see page 759)	**Edit ▶ Select All** Selects all objects in the main window. Select All works with text when using the **Code Editor** and with components when using the **Form Designer**.
Send to Back (↗ see page 759)	**Edit ▶ Send to Back** Moves selected objects to the visual back of an active form. For example: if you have a few buttons on a form in the **Form Designer**and you would like them to be visually in front of a TPanel component, you can first arrange the buttons, then position a TPanel component over the buttons and select Send to Back.
Undo (↗ see page 759)	**Edit ▶ Undo** Reverts the previous actions. Undo works in the text editor and the **Form Designer**. If you delete a character in the text editor, selecting Undo brings the character back. If you delete a component (e.g., a button) in the **Form Designer**, selecting Undo brings the component back. The undo limit is set in **Tools ▶ Options ▶ Editor Options ▶ Undo Limit**. The default is 32,767.
Redo (↗ see page 759)	**Edit ▶ Redo** Re-executes the last command that was undone using **Edit ▶ Undo**. This command is available from the **Form Designer**.

Select All Controls (see page 759)	Edit ▶Select All Controls Selects all controls in the main window.

3.2.4.1 Alignment

Edit ▶Align

Lines up selected components in relation to each other or to the form.

Item	Description
No change	Does not change the alignment of the component.
Left sides	Lines up the left edges of the selected components (horizontal only).
Centers	Lines up the centers of the selected components.
Right sides	Lines up the right edges of the selected components (horizontal only).
Tops	Lines up the top edges of the selected components (vertical only).
Bottoms	Lines up the bottom edges of the selected components (vertical only).
Space equally	Lines up the selected components equidistant from each other.
Center in window	Lines up the selected components with the center of the window.

See Also

Adding Components to a Form (see page 152)

3.2.4.2 Creation Order

Edit ▶Creation Order

Specifies the order in which your application creates nonvisual components when you load the form at design time or runtime.

The list box displays only those nonvisual components on the active form, their type, and their current creation order. The default creation order is determined by the order in which you placed the nonvisual components on the form.

3.2.4.3 Edit Tab Order

Edit▶Tab Order

Modifies the tab order of the components on the VCL Form, or within the selected component if that component contains other components.

For example: a Form containing an edit control with the Tab property value of 0, and a button with the Tab property of 1, the edit control comes before the button when you cycle through the controls with the `Tab` key. The **Tab Order** dialog facilitates changing the values of the Tab property using the **Up** and **Down** arrows instead of explicitly having to specify values for the Tab property in the **Object Inspector**.

Click the **Up** arrow to move the component up in the tab order, or click the **Down** arrow to move it down in the tab order.

3.2.4.4 Scale

Edit ▶Scale

Proportionally resizes all the components on the current form. Scaling the form repositions the components as well as resizing them.

Enter a percentage to which you want to resize the form’s components. The scaling factor must be between 25 and 400. Percentages over 100 grow the form’s components. Percentages under 100 shrink the form’s components.

See Also

Adding Components to a Form (see page 152)

3.2.4.5 Size

Edit ▶Size

Resizes multiple components to be exactly the same height or width.

Item	Description
No change	Does not change the size of the components.
Shrink to smallest	Resizes the group of components to the height or width of the smallest selected component.
Grow to largest	Resizes the group of components to the height or width of the largest selected component.
Width	Sets a custom width for the selected components.
Height	Sets a custom height for the selected components.

See Also

Adding Components to a Form (see page 152)

3.2.4.6 Align to Grid

Edit ▶Align to Grid

Aligns selected objects to the grid in the **Form Designer**. By default, components are aligned to a four-pixel grid used by the **Form Designer**. Press ALT to override this default when positioning components.

See Also

Adding Components to a Form (see page 152)

3.2.4.7 Bring to Front

Edit ▶Bring to Front

Brings selected objects to the visual front of an active form. For example: if you have are working on a form in the **Form Designer** and you drop a TPanel over a portion of a TListbox, select Bring to Front to have the TListbox lie over the TPanel.

3.2.4.8 Copy

Edit ▶Copy

Copies the current selection to the clipboard. Copy works with text when using the **Code Editor** and with components when using the **Form Designer**.

3.2.4.9 Cut

Edit ▶Cut

Removes the current selection and stores it to the clipboard. Cut works with text when using the **Code Editor** and with components when using the **Form Designer**.

3.2.4.10 Delete

Edit ▶Delete

Removes the selected object or group of objects. Delete works with text when using the **Code Editor** and with components when using the **Form Designer**.

If you accidentally delete an object, use the **Edit▶Undo** command.

3.2.4.11 Flip Children

Edit ▶Flip Children▶All

Edit ▶Flip Children▶Selected

Reverses Bi-Directional (BiDi) support. This is useful when you design a form for a language where the text reads right to left, such as Japanese or Arabic.

Item	Description
All	Horizontally reverses all the controls on a form.
Selected	Horizontally reverses all the selected controls on a form.

3.2.4.12 Lock Controls

Edit ▶Lock Controls

Locks objects in place on an active form. You cannot move locked objects. To unlock, select Lock Controls again.

3.2.4.13 Paste

Edit ▶Paste

Pastes contents of the clipboard previously caputured using Cut or Copy. Paste works with text when using the **Code Editor** and with components when using the **Form Designer**.

3.2.4.14 Select All

Edit ▶Select All

Selects all objects in the main window. Select All works with text when using the **Code Editor** and with components when using the **Form Designer**.

3.2.4.15 Send to Back

Edit ▶Send to Back

Moves selected objects to the visual back of an active form. For example: if you have a few buttons on a form in the **Form Designer**and you would like them to be visually in front of a TPanel component, you can first arrange the buttons, then position a TPanel component over the buttons and select Send to Back.

3.2.4.16 Undo

Edit▶Undo

Reverts the previous actions. Undo works in the text editor and the **Form Designer**. If you delete a character in the text editor, selecting Undo brings the character back. If you delete a component (e.g., a button) in the **Form Designer**, selecting Undo brings the component back.

The undo limit is set in **Tools▶Options▶Editor Options▶Undo Limit**. The default is 32,767.

3.2.4.17 Redo

Edit ▶Redo

Re-executes the last command that was undone using **Edit▶Undo**. This command is available from the **Form Designer**.

3.2.4.18 Select All Controls

Edit ▶Select All Controls

Selects all controls in the main window.

3.2.5 Error Messages

Topics

Name	Description
Data Breakpoint is set on a stack location (↗ see page 762)	Stack memory is volatile and changes a lot. When a data breakpoint is set on an address in the stack, the breakpoint might be hit quite often, each time something is pushed on or popped off the stack. You can still set this breakpoint, but be aware that it might get hit so often that the application cannot run properly.
Misaligned Data Breakpoint (↗ see page 762)	You attempted to set a data breakpoint whose Address is misaligned with respect to its Length. If the data breakpoint has a two-byte length, then the Address should be aligned on a two-byte boundary. Similarly, the Address of a four-byte data breakpoint should be aligned on a four-byte boundary. You can still set this data breakpoint, but if you do, the breakpoint will be set on an address that is on an appropriate boundary. This can trigger the breakpoint in unintended situations.
Error address not found (↗ see page 762)	The address you have specified cannot be mapped to a source code position. This error usually occurs for one of the following reasons: • The address you entered is invalid or is not an address in your application. • The module containing the specified address was not compiled with debug information. • The address specified does not correspond to a program statement. Note that the runtime and visual component libraries are compiled without debug information.
Another file named <FileName> is already on the search path (↗ see page 763)	A file with the same name as the one you just specified is already in another directory on the search path.
Could not stop due to hard mode (↗ see page 763)	The integrated debugger has detected that Windows is in a modal state and will not allow the debugger to stop your application. Windows enters "hard mode" whenever processing an inter-task SendMessage, when there is no task queue, or when the menu system is active. You will not generally encounter hard mode unless you are debugging DDE or OLE processes within Delphi. A standalone debugger such as the Turbo Debugger for Windows can be used to debug applications even when Windows is in hard mode.
Error creating process: <Process> (<ErrorCode>) (↗ see page 763)	Delphi was unable to start your application for the reason specified. For more information about "Insufficient memory to run" errors, see the README.TXT file.
A component class named <name> already exists (↗ see page 763)	A package with the name you specified is already installed in the IDE. Rename the package or check if the package you are trying to install is already there.
A field or method named <name> already exists (↗ see page 763)	The name you have specified is already being used by an existing method or field. For a complete list of all fields and methods defined, check the form declaration at the top of the unit source file.
The project already contains a form or module named <Name> (↗ see page 763)	Every module name (program or library, form, and unit) in a project must be unique.
Incorrect field declaration in class <ClassName> (↗ see page 763)	In order to keep your form and source code synchronized, Delphi must be able to find and maintain the declaration of each field in the first section of the form's class definition. Though the compiler allows more complex syntax, the form Designer will report an error unless each field that is declared in this section is equivalent to the following:
Field <Field Name> does not have a corresponding component. Remove the declaration? (↗ see page 764)	The first section of your form's class declaration defines a field for which there is no corresponding component on the form. Note that this section is reserved for use by the form Designer. To declare your own fields and methods, place them in a separate public, private, or protected section. This error will also occur if you load the binary form file (.DFM) into the Code editor and delete or rename one or more components.

Field <Field Name> should be of type <Type1> but is declared as <Type2>. Correct the declaration? (see page 764)	The type of specified field does not match its corresponding component on the form. This error will occur if you change the field declaration in the Code editor or load the binary form file (.DFM) into the Code editor and modify the type of a component. If you select No and run your application, an error will occur when the form is loaded.
Declaration of class <ClassName> is missing or incorrect (see page 764)	Delphi is unable to locate the form's class declaration in the interface section of the unit. This is probably because the type declaration containing the class has been deleted, commented out, or incorrectly modified. This error will occur if Delphi cannot locate a class declaration equivalent to the following:
Module header is missing or incorrect (see page 764)	The module header has been deleted, commented out, or otherwise incorrectly modified. Use UNDO to reverse your changes, or correct the declaration manually. In order to keep your form and source code synchronized, Delphi must be able to find a valid module header at the beginning of the source file. A valid module header consists of the reserved word unit, program or library, followed by an identifier (for example, Unit1, Project1), followed by a semi-colon. The file name must match the identifier. For example, Delphi will look for a unit named Unit1 in UNIT1.PAS, a project named Project1 in PROJECT1.DPR,... more (see page 764)
IMPLEMENTATION part is missing or incorrect (see page 765)	In order to keep your form and source code synchronized, Delphi must be able to find the unit's implementation section. This reserved word has been deleted, commented out, or misspelled. Use UNDO to reverse your changes or correct the reserved word manually.
Insufficient memory to run (see page 765)	Delphi was unable to run your application due to insufficient memory or Windows resources. Close other Windows applications and try again. This error sometimes occurs because of insufficient low (conventional) memory. For further information, see the README.TXT file.
Breakpoint is set on line that contains no code or debug information. Run anyway? (see page 765)	A breakpoint is set on a line that does not generate code or in a module which is not part of the project. If you choose to run anyway, invalid breakpoints will be disabled (ignored).
<IDname> is not a valid identifier (see page 765)	The identifier name is invalid. Ensure that the first character is a letter or an underscore (_). The characters that follow must be letters, digits, or underscores, and there cannot be any spaces in the identifier.
<Library Name>is already loaded, probably as a result of an incorrect program termination. Your system may be unstable and you should exit and restart Windows now. (see page 765)	An error occurred while attempting to initialize Delphi's component library. One or more DLLs are already in memory, probably as a result of an incorrect program termination in a previous Delphi or BDE session. You should exit and then restart Windows.
Incorrect method declaration in class <ClassName> (see page 765)	In order to keep your form and source code synchronized, Delphi must be able to find and maintain the declaration of each method in the first section of the form's class definition. The form Designer will report an error unless the field and method declarations in this section are equivalent to the following:
Cannot find implementation of method <MethodName> (see page 766)	The indicated method is declared in the form's class declaration but cannot be located in the implementation section of the unit. It probably has been deleted, commented out, renamed, or incorrectly modified. Use UNDO to reverse your changes, or correct the procedure declaration manually. Be sure the declaration in the class is identical to the one in the implementation section. (This is done automatically if you use the Object Inspector to create and rename event handlers.)
The <Method Name> method referenced by <Form Name>.<Event Name> has an incompatible parameter list. Remove the reference? (see page 766)	A form has been loaded that contains an event property mapped to a method with an incompatible parameter list. Parameter lists are incompatible if the number or types of parameters are not identical. For a list of methods declared in this form which are compatible for this event property, use the dropdown list on the Object Inspector's Events page. This error occurs when you manually modify a method declaration that is referenced by an event property. Note that it is unsafe to run this program without removing the reference or correcting the error.
The <Method Name> method referenced by <Form Name> does not exist. Remove the reference? (see page 766)	The indicated method is no longer present in the class declaration of the form. This error occurs when you manually delete or rename a method in the form's class declaration that is assigned to an event property. If you select No and run this application, an error will occur when the form is loaded.
No code was generated for the current line (see page 766)	You are attempting to run to the cursor position, but you have specified a line that did not generate code, or is in a module which is not part of the project. Specify another line and try again. Note that the smart linker will remove procedures that are declared but not called by the program (unless they are virtual method of an object that is linked in).

Property and method <MethodName> are not compatible (↗ see page 767)	You are assigning a method to an event property even though they have incompatible parameter lists. Parameter lists are incompatible if the number of types of parameters are not identical. For a list of compatible methods in this form, see the dropdown list on the Object Inspector Events page.
Cannot find <FileName.PAS> or <FileName.DCU> on the current search path (↗ see page 767)	The .pas or .dcu file you just specified cannot be found on the search path. You can modify the search path, copy the file to a directory along the path, or remove the file from the list of installed units.
Source has been modified. Rebuild? (↗ see page 767)	You have made changes to one or more source or form modules while your application is running. When possible, you should terminate your application normally (select No, switch to your running application, and select Close on the System Menu), and then run or compile again. If you select Yes, your application will be terminated and then recompiled.
Symbol <BrowseSymbol> not found. (↗ see page 767)	The browser cannot find the specified symbol. This error occurs if you enter an invalid symbol name or if debug information is not available for the module that contains the specified symbol.
Debug session in progress. Terminate? (↗ see page 767)	Your application is running and will be terminated if you proceed. When possible, you should cancel this dialog and terminate your application normally (for example, by selecting Close on the System Menu).
Uses clause is missing or incorrect (↗ see page 767)	In order to keep your forms and source code synchronized, Delphi must be able to find and maintain the **uses** clause of each module. In a unit, a valid **uses** clause must be present immediately following the interface reserved word. In a program or library, a valid **uses** clause must be present immediately following the program or library header. This error occurs because the **uses** clause has been deleted, commented out, or incorrectly modified. Use undo to reverse your changes or correct the declaration manually.
Invalid event profile <Name> (↗ see page 768)	The VBX control you are installing is invalid.

3.2.5.1 Data Breakpoint is set on a stack location

Stack memory is volatile and changes a lot. When a data breakpoint is set on an address in the stack, the breakpoint might be hit quite often, each time something is pushed on or popped off the stack.

You can still set this breakpoint, but be aware that it might get hit so often that the application cannot run properly.

3.2.5.2 Misaligned Data Breakpoint

You attempted to set a data breakpoint whose Address is misaligned with respect to its Length.

If the data breakpoint has a two-byte length, then the Address should be aligned on a two-byte boundary. Similarly, the Address of a four-byte data breakpoint should be aligned on a four-byte boundary.

You can still set this data breakpoint, but if you do, the breakpoint will be set on an address that is on an appropriate boundary. This can trigger the breakpoint in unintended situations.

3.2.5.3 Error address not found

The address you have specified cannot be mapped to a source code position. This error usually occurs for one of the following reasons:

- The address you entered is invalid or is not an address in your application.
- The module containing the specified address was not compiled with debug information.
- The address specified does not correspond to a program statement.

Note that the runtime and visual component libraries are compiled without debug information.

3.2.5.4 Another file named <FileName> is already on the search path

A file with the same name as the one you just specified is already in another directory on the search path.

3.2.5.5 Could not stop due to hard mode

The integrated debugger has detected that Windows is in a modal state and will not allow the debugger to stop your application. Windows enters "hard mode" whenever processing an inter-task SendMessage, when there is no task queue, or when the menu system is active. You will not generally encounter hard mode unless you are debugging DDE or OLE processes within Delphi.

A standalone debugger such as the Turbo Debugger for Windows can be used to debug applications even when Windows is in hard mode.

3.2.5.6 Error creating process: <Process> (<ErrorCode>)

Delphi was unable to start your application for the reason specified.

For more information about "Insufficient memory to run" errors, see the README.TXT file.

3.2.5.7 A component class named <name> already exists

A package with the name you specified is already installed in the IDE. Rename the package or check if the package you are trying to install is already there.

3.2.5.8 A field or method named <name> already exists

The name you have specified is already being used by an existing method or field.

For a complete list of all fields and methods defined, check the form declaration at the top of the unit source file.

3.2.5.9 The project already contains a form or module named <Name>

Every module name (program or library, form, and unit) in a project must be unique.

3.2.5.10 Incorrect field declaration in class <ClassName>

In order to keep your form and source code synchronized, Delphi must be able to find and maintain the declaration of each field in the first section of the form's class definition. Though the compiler allows more complex syntax, the form Designer will report an error unless each field that is declared in this section is equivalent to the following:

```
type
       ...
       TForm1 = class(TForm)
       Field1:FieldType;
```

```
        Field2:FieldType;
        ...
```

This error has occurred because one or more declarations in this section have been deleted, commented out, or incorrectly modified. Use undo to reverse your changes or correct the declaration manually.

Note that this first section of the form's class declaration is reserved for use by the form Designer. To declare your own fields and methods, place them in a separate public, private, or protected section.

3.2.5.11 Field <Field Name> does not have a corresponding component. Remove the declaration?

The first section of your form's class declaration defines a field for which there is no corresponding component on the form. Note that this section is reserved for use by the form Designer.

To declare your own fields and methods, place them in a separate public, private, or protected section.

This error will also occur if you load the binary form file (.DFM) into the Code editor and delete or rename one or more components.

3.2.5.12 Field <Field Name> should be of type <Type1> but is declared as <Type2>. Correct the declaration?

The type of specified field does not match its corresponding component on the form. This error will occur if you change the field declaration in the Code editor or load the binary form file (.DFM) into the Code editor and modify the type of a component.

If you select No and run your application, an error will occur when the form is loaded.

3.2.5.13 Declaration of class <ClassName> is missing or incorrect

Delphi is unable to locate the form's class declaration in the interface section of the unit. This is probably because the type declaration containing the class has been deleted, commented out, or incorrectly modified. This error will occur if Delphi cannot locate a class declaration equivalent to the following:

```
type
        ...
        TForm1 = class(TForm)
        ...
```

Use UNDO to reverse your edits, or correct the declaration manually.

3.2.5.14 Module header is missing or incorrect

The module header has been deleted, commented out, or otherwise incorrectly modified. Use UNDO to reverse your changes, or correct the declaration manually.

In order to keep your form and source code synchronized, Delphi must be able to find a valid module header at the beginning of the source file. A valid module header consists of the reserved word unit, program or library, followed by an identifier (for example, Unit1, Project1), followed by a semi-colon. The file name must match the identifier.

For example, Delphi will look for a unit named Unit1 in UNIT1.PAS, a project named Project1 in PROJECT1.DPR, and a library (.DLL) named MyDLL in MYDLL.DPR.

Note that module identifiers cannot exceed eight characters in length.

3.2.5.15 IMPLEMENTATION part is missing or incorrect

In order to keep your form and source code synchronized, Delphi must be able to find the unit's implementation section. This reserved word has been deleted, commented out, or misspelled.

Use UNDO to reverse your changes or correct the reserved word manually.

3.2.5.16 Insufficient memory to run

Delphi was unable to run your application due to insufficient memory or Windows resources. Close other Windows applications and try again.

This error sometimes occurs because of insufficient low (conventional) memory. For further information, see the README.TXT file.

3.2.5.17 Breakpoint is set on line that contains no code or debug information. Run anyway?

A breakpoint is set on a line that does not generate code or in a module which is not part of the project. If you choose to run anyway, invalid breakpoints will be disabled (ignored).

3.2.5.18 <IDname> is not a valid identifier

The identifier name is invalid. Ensure that the first character is a letter or an underscore (_). The characters that follow must be letters, digits, or underscores, and there cannot be any spaces in the identifier.

3.2.5.19 <Library Name>is already loaded, probably as a result of an incorrect program termination. Your system may be unstable and you should exit and restart Windows now.

An error occurred while attempting to initialize Delphi's component library. One or more DLLs are already in memory, probably as a result of an incorrect program termination in a previous Delphi or BDE session.

You should exit and then restart Windows.

3.2.5.20 Incorrect method declaration in class <ClassName>

In order to keep your form and source code synchronized, Delphi must be able to find and maintain the declaration of each method in the first section of the form's class definition. The form Designer will report an error unless the field and method declarations in this section are equivalent to the following:

```
type
       ...
```

```
        TForm1 = class(TForm)
        Field1:FieldType;
        Field2:FieldType;
        ...
        <Method1 Declaration>;
        <Method2 Declaration>;
        ...
    ...
```

This error has occurred because one or more method declarations in this section have been deleted, commented out, or incorrectly modified. Use undo to reserve your changes or correct the declaration manually.

Note that this first section of the form's class declaration is reserved for use by the form Designer. To declare your own fields and methods, place them in a separate public, private, or protected section.

3.2.5.21 Cannot find implementation of method <MethodName>

The indicated method is declared in the form's class declaration but cannot be located in the implementation section of the unit. It probably has been deleted, commented out, renamed, or incorrectly modified.

Use UNDO to reverse your changes, or correct the procedure declaration manually. Be sure the declaration in the class is identical to the one in the implementation section. (This is done automatically if you use the Object Inspector to create and rename event handlers.)

3.2.5.22 The <Method Name> method referenced by <Form Name>.<Event Name> has an incompatible parameter list. Remove the reference?

A form has been loaded that contains an event property mapped to a method with an incompatible parameter list. Parameter lists are incompatible if the number or types of parameters are not identical.

For a list of methods declared in this form which are compatible for this event property, use the dropdown list on the Object Inspector's Events page.

This error occurs when you manually modify a method declaration that is referenced by an event property.

Note that it is unsafe to run this program without removing the reference or correcting the error.

3.2.5.23 The <Method Name> method referenced by <Form Name> does not exist. Remove the reference?

The indicated method is no longer present in the class declaration of the form. This error occurs when you manually delete or rename a method in the form's class declaration that is assigned to an event property.

If you select No and run this application, an error will occur when the form is loaded.

3.2.5.24 No code was generated for the current line

You are attempting to run to the cursor position, but you have specified a line that did not generate code, or is in a module which is not part of the project.

Specify another line and try again.

Note that the smart linker will remove procedures that are declared but not called by the program (unless they are virtual method of an object that is linked in).

3.2.5.25 Property and method <MethodName> are not compatible

You are assigning a method to an event property even though they have incompatible parameter lists. Parameter lists are incompatible if the number of types of parameters are not identical. For a list of compatible methods in this form, see the dropdown list on the Object Inspector Events page.

3.2.5.26 Cannot find <FileName.PAS> or <FileName.DCU> on the current search path

The .pas or .dcu file you just specified cannot be found on the search path.

You can modify the search path, copy the file to a directory along the path, or remove the file from the list of installed units.

3.2.5.27 Source has been modified. Rebuild?

You have made changes to one or more source or form modules while your application is running. When possible, you should terminate your application normally (select No, switch to your running application, and select Close on the System Menu), and then run or compile again.

If you select Yes, your application will be terminated and then recompiled.

3.2.5.28 Symbol <BrowseSymbol> not found.

The browser cannot find the specified symbol. This error occurs if you enter an invalid symbol name or if debug information is not available for the module that contains the specified symbol.

3.2.5.29 Debug session in progress. Terminate?

Your application is running and will be terminated if you proceed. When possible, you should cancel this dialog and terminate your application normally (for example, by selecting Close on the System Menu).

3.2.5.30 Uses clause is missing or incorrect

In order to keep your forms and source code synchronized, Delphi must be able to find and maintain the **uses** clause of each module.

In a unit, a valid **uses** clause must be present immediately following the interface reserved word. In a program or library, a valid **uses** clause must be present immediately following the program or library header.

This error occurs because the **uses** clause has been deleted, commented out, or incorrectly modified. Use undo to reverse your changes or correct the declaration manually.

3.2.5.31 Invalid event profile <Name>

The VBX control you are installing is invalid.

3.2.6 File

Topics

Name	Description
Active Form Wizard (↗ see page 771)	**File ▶ New ▶ Other... ▶ Active Form** Use the Active Form wizard to add an Active Form to an ActiveX Library project. The wizard creates an ActiveX Library project (if needed), a type library, a form, an implementation unit, and a unit containing corresponding type library declarations.
Active Server Object wizard (↗ see page 772)	**File ▶ New ▶ Other ▶ Active Server Object** Use the Active Server Object wizard to create a simple active server object. Before you create an Active Server Object, create or open the project for an application containing functionality that you want to implement. The project can be either an application or ActiveX library, depending on your needs. In the dialog, specify the properties of your Active Server Object, which is a special Automation object created by and called from the script running in an Active Server Page.
Add (↗ see page 773)	Use this dialog box to add a package to the Requires clause in the current package.
Automation Object Wizard (↗ see page 773)	**File ▶ New ▶ Other...** Use the **New Automation Object** wizard to add an Automation server to an ActiveX Library project. The wizard creates a type library, and the definition for the Automation object.
Browse With Dialog box (↗ see page 774)	Use this dialog box to maintain a list of external browsers and specify which browser to use by default.
Browse With Dialog box (↗ see page 775)	Use this dialog box to specify the properties for the selected external browser.
COM Object Wizard (↗ see page 775)	**File ▶ New ▶ Other...** Use the **New COM Object** wizard to create a simple COM object such as a shell extension. Before you create a COM object, create or open the project for an application containing functionality that you want to implement. The project can be either an application or ActiveX library, depending on your needs.
COM+ Event Interface Selection dialog box (↗ see page 776)	On the COM+ Subscription Object wizard, use the Browse button to display the COM+ Event Interface Selection dialog box. It lists all event classes currently installed in the COM+ Catalog. The dialog also contains a Browse button that you can use to search for and select a type library containing the event interface.
COM+ Event Object Wizard (↗ see page 776)	**File ▶ New ▶ Other...** The **COM+ Event Object** wizard creates a COM+ event object that can be called by a transactional server to generate events on clients. Because the project for a COM+ object can only contain other COM+ objects, you may be prompted to start a new project when you launch this wizard.
COM+ Subscription Object Wizard (↗ see page 776)	**File ▶ New ▶ Other ▶ COM+ Subscription Object** You can create the COM+ event subscriber component using the COM+ Subscription Object wizard. You use this with a COM+ Event Object to receive notification of events fired by COM+ publisher applications.
Customize New Menu (↗ see page 777)	**File ▶ New ▶ Customize** Use this dialog box to customize the content of the **File ▶ New** menu by dragging menu items from the center pane and dropping them on the right pane.
Change Destination File Name (↗ see page 777)	Use this dialog box to rename the selected file when it is copied to the destination directory. The rename will not occur until you copy the file to the destination directory. The source file will not be renamed. If the file already exists in the destination directory with its original name, that file will remain in the destination directory until you delete it.
FTP Connection Options (↗ see page 778)	Use this dialog box to specify FTP server connection information for the Deployment Manager.

Interface Selection Wizard (see page 778)	**File ▶ New ▶ Other... ▶ New COM Object** The **Interface Selection** wizard is accessed by clicking the List button on the **New COM Object** wizard. The **Interface Selection** wizard lets you select a predefined dual or custom interface that you want to implement with a COM object you are creating. The selected interface becomes the default interface of the newly created COM object. The COM object wizard adds skeletal method implementations for all the methods on this interface to the generated implementation unit. You can then fill in the bodies of these methods to provide an implementation of the interface. **Warning:** The... more (see page 778)
New ASP.NET Application (see page 779)	**File ▶ New ▶ Other ▶ ASP NET Application** Use this dialog box to set options for new Web Forms applications.
New ASP.NET Content Page (see page 779)	**File ▶ New ▶ Other ▶ Delphi for .NET Projects ▶ ASP NET Content Page** Use this dialog box to create a new Content Page for an ASP.NET application.
New ASP.NET Generic Handler (see page 779)	**File ▶ New ▶ Other ▶ Delphi for .NET Projects ▶ Generic Handler** Use this dialog box to create a new HTTP Generic Handler for an ASP.NET application.
New ASP.NET Master Page (see page 780)	**File ▶ New ▶ Other ▶ Delphi for .NET Projects ▶ ASP NET Master Page** Use this dialog box to create a new Master Page for an ASP.NET application.
New ASP.NET Master Page (see page 780)	**File ▶ New ▶ Other ▶ Delphi for .NET Projects ▶ ASP NET Page** Use this dialog box to create a new Page for an ASP.NET application.
New ASP.NET User Control (see page 780)	**File ▶ New ▶ Other ▶ Delphi for .NET Projects ▶ ASP NET User Control** Use this dialog box to create a new User Control for an ASP.NET application.
New ASP.NET Web Service (see page 780)	**File ▶ New ▶ Other ▶ Delphi for .NET Projects ▶ ASP NET Web Service** Use this dialog box to create a new Web Service for an ASP.NET application.
New Console Application (see page 780)	**File ▶ New ▶ Other** Use this dialog box to create an application that runs in a console window.
New DBWeb Control Wizard (see page 781)	**File ▶ New ▶ Other ▶ Delphi ASP Projects ▶ DBWeb Control Library** Use this dialog box to create a data aware WebControl. This DB Web Control can suppplement the DB Web Controls provided on the **Tool Palette**.
New Dynamic-link Library (see page 781)	**File ▶ New ▶ Other** Use this dialog box to create a DLL project. A dynamic-link-library is a module of compiled code that provides functionality for applications.
New Items (see page 781)	**File ▶ New ▶ Other** Use this dialog box to create a new project or other entity. The **New Items** dialog box displays project templates that are stored in the RAD Studio Object Repository.
New Application (see page 782)	**File ▶ New** Use this dialog box to specify a name and location for the new application.
New Remote Data Module Object (see page 782)	**File ▶ New ▶ Other** Use this dialog box to create a data module that can be accessed remotely as a dual-interface Automation server.
New Thread Object (see page 783)	**File ▶ New ▶ Other ▶ Delphi Projects ▶ Delphi Files ▶ Thread Object** Use this dialog box to define a thread class that encapsulates a single execution thread in a multi-threaded application.
Open (see page 784)	**File ▶ Open** Use this dialog box to locate and open a file. The title of this dialog box varies, depending on the function being performed.
Package (see page 784)	**File ▶ New ▶ Other ▶ Delphi for .NET Projects ▶ Package** Use this dialog box to create a package.
Print Selection (see page 785)	**File ▶ Print** Use this dialog box to print the current file.
Project Upgrade (see page 785)	**File ▶ Open** Use this dialog box to upgrade an older Delphi project that has no corresponding `.bdsproj` project file. When you upgrade the project, the `.bdsproj` file and other files and directories used by RAD Studio will be created in the project's directory.

Project Updated (↗ see page 785)	**File ▶ Open** This dialog box appears when a project from a previous version is automatically updated for RAD Studio. The following changes are made to the project: • The encoding of project data is updated. • References to `.lib`s, `.bpi`s, and `.csm`s are updated as needed. **Note:** For compatibility issues, see the release notes.
Remote Data Module Wizard (↗ see page 786)	**File ▶ New ▶ Other ▶ Remote Data Module** Use the Remote Data Module wizard to create a data module that can be accessed remotely as a dual-interface Automation server. A remote data module resides in the application server between a client and server in a multi-tiered database environment.
Satellite Assembly Wizard (↗ see page 787)	**File ▶ New ▶ Other ▶ Delphi for .NET Projects ▶ Satellite Assembly Wizard** Use this wizard to add one or more satellite assemblies to a project. Follow the instructions on each wizard page.
Revert to Previous Revision (↗ see page 787)	This confirmation message appears when you click **Revert to previous revision** on the History tab of the Create Project dialog box.
Add New WebService (↗ see page 787)	**File ▶ New ▶ Other ▶ Delphi Projects ▶ Web Services ▶ SOAP Server Interface** Use this dialog box to define a new invokable interface and its implementation class. The dialog generates a new unit that declares an invokable interface and the implementation class. The interface descends from IInvokable, and the implementation class from TInvokableClass. It also generates the code to register the interface and implementation class with the invocation registry. After exiting the wizard, edit the generated interface and class definitions, adding in the properties and methods you want to expose as your Web Service.
SOAP Data Module Wizard (↗ see page 788)	**File ▶ New ▶ Other ▶ Delphi Projects ▶ Web Services ▶ SOAP Server Data Module** Use this dialog box to add a SOAP data module to a Web Service application. A SOAP data module allows a Web Service application to export database information as a Web Service. Client datasets on the client application can display and update this database information.
New SOAP Server Application (↗ see page 788)	**File ▶ New ▶ Other ▶ Delphi Projects ▶ Web Services ▶ SOAP Server Application** Use this dialog box to specify the type of server your Web Service application will work with.
Save As (↗ see page 788)	**File ▶ Save As** Use this dialog box to to save the active file.
Select Directory (↗ see page 789)	Use this dialog box to to choose a working directory for your new project.
Transactional Object Wizard (↗ see page 789)	**File ▶ New ▶ Other...** Use the New Transactional Object wizard to create a server object that runs under MTS or COM+. Transactional objects are used in distributed applications to make use of the special services supplied by MTS or COM+ for resource management, transaction support, or security.
Use Unit (↗ see page 790)	**File ▶ Use Unit** Use this dialog box to make the contents of a unit available to the current unit. This dialog lists of all units in the project that are not currently used or included by the current unit. You can only use units that are part of the current project.
WSDL Import Options (↗ see page 791)	Use this dialog box to specify information that the importer needs to connect to the server that publishes a WSDL document or to configure the way the wizard generates code to represent the definitions in a WSDL document.
WSDL Import Wizard (↗ see page 792)	**File ▶ New ▶ Other ▶ Delphi Projects ▶ Web Services ▶ WSDL Importer** Use this wizard to import a WSDL document or XML schema that describes a Web Service. Once you have imported the WSDL document or XML schema, the wizard generates all the interface and class definitions you need for calling on those Web Services using a remote interfaced object (THTTPRIO). You can also tell the wizard to generate skeletal code you can complete to create a Web Service application (for example, if you want to implement a Web Service that is already defined in a WSDL document).
New Web Server Application (↗ see page 793)	**File ▶ New ▶ Other ▶ Delphi Projects ▶ New ▶ Web Server Application** Use this dialog box to specify the type of server your Web server application will work with.

Add New WebService (↗ see page 793)	**File ▶ New ▶ Other ▶ Delphi Projects ▶ WebServices ▶ SOAP Server Interface** Use this dialog box to generate a new unit that declares a invokable interface and its implementation class. The interface descends from IInvokable, and the implementation class from TInvokableClass. It also generates the code to register the interface and implementation class with the invocation registry. After exiting the wizard, edit the generated interface and class definitions, adding in the properties and methods you want to expose as your Web Service.
Application Module Page Options/New WebSnap Page Module (↗ see page 794)	Use this dialog box to define the basic properties of a page module. The dialog title varies based on how you accessed the dialog.
New WebSnap Application (↗ see page 795)	**File ▶ New ▶ Other ▶ Delphi Projects ▶ WebSnap ▶ WebSnap Application** Use this dialog box to configure a new WebSnap application.
New WebSnap Data Module (↗ see page 796)	Use this dialog box to specify how the server handles the creation and destruction of your data module.
Web App Components (↗ see page 796)	Use this dialog box to select component categories and to select specific components in each of the following categories (some categories offer only one choice).
XML Data Binding Wizard Options (↗ see page 797)	**File ▶ New ▶ Other ▶ Delphi Project ▶ New ▶ XML Data Binding ▶ Options button** Use this dialog box to determine how the **XML Data Binding Wizard** generates interfaces and implementation classes to represent an XML document or schema.
XML Data Binding Wizard, page 1 (↗ see page 797)	**File ▶ New ▶ Other ▶ Delphi Project ▶ New ▶ XML Data Binding** Use this wizard to generate interface and class definitions that correspond to the structure of an XML document or schema. The wizard generates a global function that returns the interface for the root element of the document. After you use the wizard to create these definitions, you can use the classes and interfaces to work with XML documents that have the structure of the specified document or schema.
XML Data Binding Wizard, page 2 (↗ see page 798)	**File ▶ New ▶ Other ▶ Delphi Project ▶ New ▶ XML Data Binding** Use this wizard page to specify what code the wizard generates.
XML Data Binding Wizard, page 3 (↗ see page 799)	**File ▶ New ▶ Other ▶ Delphi Project ▶ New ▶ XML Data Binding** Use this wizard page to confirm the choices you have made, specify unit-wide code generation options, indicate where you want your choices saved, and tell the wizard to generate code to represent the XML document or schema.
Close (↗ see page 799)	**File ▶ Close** **File ▶ Close All** Closes the current open project or all the open projects.
Exit (↗ see page 799)	**File ▶ Exit** Closes the IDE and all open projects and files.
New (↗ see page 800)	**File ▶ New** Creates a project of the selected type.

3.2.6.1 Active Form Wizard

File ▶ New ▶ Other... ▶ Active Form

Use the Active Form wizard to add an Active Form to an ActiveX Library project. The wizard creates an ActiveX Library project (if needed), a type library, a form, an implementation unit, and a unit containing corresponding type library declarations.

Item	Description
VCL Class Name	When creating Active forms, this control is disabled because active forms are always based on TActiveForm.
New ActiveX Name	The wizard provides a default name that clients will use to identify your Active form. Change this name to provide a different OLE class name.
Implementation Unit	The wizard a default name for the unit that contains.CPP and .H files that contain the code that implements the behavior of the Active form. You can accept the default name or type in a new name.

Project Name	Active forms must be added to an ActiveX library project. If you currently don't have an ActiveX Library project open, a fourth field allows you to specify which ActiveX Library project to add the ActiveForm control. A default Project Name is provided. This control is disabled if you have an ActiveX Library open..
Threading Model	Choose the threading model to indicate how COM serializes calls to your ActiveX form. Note: The threading model you choose determines how the object is registered. You must make sure that your object implementation adheres to the model selected.
Include Version Information	This option includes version information in the .OCX file. Adding this resource to your control allows your control to expose information about the module, such as copyright and file description, which can be viewed in the browser. Version information can be specified by choosing **Project ▶ Options** and selecting the **Version Info** page.
Include About Box	When this box is checked, an About box is included in the project. The About box is a separate form that you can modify. By default, the About box includes the name of the Active Form, an image, copyright information, and an OK button.

3.2.6.2 Active Server Object wizard

File ▶ New ▶ Other ▶ Active Server Object

Use the Active Server Object wizard to create a simple active server object. Before you create an Active Server Object, create or open the project for an application containing functionality that you want to implement. The project can be either an application or ActiveX library, depending on your needs.

In the dialog, specify the properties of your Active Server Object, which is a special Automation object created by and called from the script running in an Active Server Page.

Item	**Description**
CoClass Name	Specify the name for the object that you want to implement. This is the CoClass name that appears in the type library. The generated implementation class has the same name with a ‘T’ prepended.
Instancing	Specify an instancing mode to indicate how your Active server is launched. (This value is ignored for in-process servers.)
Instancing	Meaning
Internal	The object can only be created internally. An external application cannot create an instance of the object directly.
Single Instance	Allows only a single COM interface for each executable (application), so creating multiple instances results in launching multiple instances of the application.
Multiple Instance	Specifies that multiple clients can connect to the application. Any time a client requests the object, a separate instance is created within a single process space. (That is, there can be multiple instances in a single executable.)
Threading Model	Choose the threading model to indicate how COM serializes calls to your active server object’s interface. The threading model you choose determines how the object is registered. You must make sure that your object implementation adheres to the model selected. Active server objects can use the following threading models:
Model	Description
Single	Only one client thread can be serviced at a time. COM serializes all incoming calls to enforce this. Your code needs no thread support.

Apartment	Each object instantiated by a client is accessed by one thread at a time. You must protect against multiple threads accessing global memory, but objects can safely access their own instance data (object properties and members).
Free	Each object instance may be called by multiple threads simultaneously. You must protect instance data as well as global memory.
Both	This is the same as the Free-threaded model, except that all callbacks supplied by clients are guaranteed to execute in the same thread. This means you do not need protect values supplied as parameters to callback functions.
Neutral	Multiple clients can call the object on different threads at the same time, but COM ensures that no two calls conflict. You must guard against thread conflicts involving global data and any instance data that is accessed by more than one method. This model should not be used with objects that have a user interface. This model is only available under COM+. Under COM, it is mapped to the Apartment model.
Page-level event methods (OnStartPage/OnEndPage)	Creates an active server object that implements OnStartPage and OnEndPage. These methods are called by the web server on initialization and finalization of the page. This style of active server objects is available for use with IIS 3 and IIS 4. Active server objects used by IIS 5 should be created using the Object Context option.
Object Context	Creates an active server object that uses MTS or COM+ to retrieve the correct instance data of your object. Recommended for use with IIS 5 (may also work with IIS 4 and MTS).
Generate a template test script for this object	Optionally provide a pop-up link to a topic for the first of an unfamiliar term in a Help topic.
Item G	Generates a simple .ASP page that creates the Active Server Object based on its ProgID. You can then edit this Active Server page to call the methods of your object.

3.2.6.3 Add

Use this dialog box to add a package to the Requires clause in the current package.

Item	Description
Package name	Enter the name of the package to add. If the package is in the **Search Path**, a full path name is not required. (If the package directory is not in the **Search Path**, it will be added to the end.)
Search Path	If you haven't included a full directory path in the **Package Name** edit box (see above), make sure the directory where your package resides is in this list. If you add a directory in the **Search Path** edit box, you will be changing the global Library Search Path. When a Delphi package is required by another package, the product must find the package's .dcpil file in order to compile.

3.2.6.4 Automation Object Wizard

File ▶New▶Other...

Use the **New Automation Object** wizard to add an Automation server to an ActiveX Library project. The wizard creates a type library, and the definition for the Automation object.

Item	Description
CoClass Name	Specify the class whose properties and methods you want to expose to client applications. (Delphi prepends a T to this name.)

Instancing	Specify an instancing mode to indicate how your Automation server is launched.
Threading Model	Choose the threading model to indicate how COM serializes calls to your Automation object's interface. The threading model you choose determines how the object is registered. You must make sure that your object implementation adheres to the model selected.
Generate Event support code	Check this box to tell the wizard to implement a separate interface for managing events on your Automation object. The separate interface has the name ICoClassNameEvents, and defines the event handlers that must be implemented by the client. Your application does not implement this interface.

The **Instancing** dropdown list can have any of the following instancing types:

Instancing	Meaning
Internal	The object can only be created internally. An external application cannot create an instance of the object directly.
Single Instance	Allows only a single COM interface for each executable (application), so creating multiple instances results in launching multiple instances of the application.
Multiple Instance	Specifies that multiple clients can connect to the application. Any time a client requests the object, a separate instance is created within a single process space. (That is, there can be multiple instances in a single executable.)

Note: Under COM+, the serialization of calls to your object is also influenced by how it participates in activities. This can be configured using the COM+ page of the type library editor or the COM+ Component Manager.

The Threading Model dropdown list can have any of the following values:

Threading Model	Meaning
Single Apartment	Only one client thread can be serviced at a time. COM serializes all incoming calls to enforce this. Your code needs no thread support.
Free	Each object instance may be called by multiple threads simultaneously. You must protect instance data as well as global memory.
Both	This is the same as the Free-threaded model, except that all callbacks supplied by clients are guaranteed to execute in the same thread. This means you do not need protect values supplied as parameters to callback functions.
Neutral	Multiple clients can call the object on different threads at the same time, but COM ensures that no two calls conflict. You must guard against thread conflicts involving global data and any instance data that is accessed by more than one method. This model should not be used with objects that have a user interface. This model is only available under COM+. Under COM, it is mapped to the Apartment model.

Note: Under COM+, the serialization of calls to your object is also influenced by how it participates in activities. This can be configured using the COM+ page of the type library editor or the COM+ Component Manager.

3

3.2.6.5 Browse With Dialog box

Use this dialog box to maintain a list of external browsers and specify which browser to use by default.

Item	Description
Browser List	Displays the list of available browsers.
Add, Delete, Edit	Enable you to add or delete browsers, and edit information selected browsers in the list.

Set as Default	Specifies which external browser to use by default for viewing web pages and web services.

3.2.6.6 Browse With Dialog box

Use this dialog box to specify the properties for the selected external browser.

Item	Description
Program	Displays path and name of the executable for the browser.
Title	Specifies the name of the browser as you want it to appear in the **Browser List** in the **Browse With** dialog box.

3.2.6.7 COM Object Wizard

File ▶New▶Other...

Use the **New COM Object** wizard to create a simple COM object such as a shell extension. Before you create a COM object, create or open the project for an application containing functionality that you want to implement. The project can be either an application or ActiveX library, depending on your needs.

Item	Description
CoClass Name	Specify the class whose properties and methods you want to expose to client applications. This is the name of the CoClass. The implementation class has the same name with a T prepended.
Instancing	Specify an instancing mode to indicate how your COM object is launched. When your COM object is used only as an in-process server, instancing is ignored.
Threading Model	Choose the threading model to indicate how client applications can call your COM object's interface. The threading model you choose determines how the object is registered. You must make sure that your object implementation adheres to the model selected.
Implemented Interface	Indicates the name of the COM object's default interface. By default, the COM object's interface is the same as the CoClass name, with an 'I' prepended. When you accept the default interface, your object gets a new interface that descends from IUnknown, which you can then define using the Type Library editor. You can change the default name of the interface by typing a name into the edit box. Instead of implementing a new interface, you can choose to have your object implement any dual or custom interface that is in a type library registered on your system. To choose the interface to implement, click the **List** button, which displays the **Interface Selection** wizard. Note that this wizard takes a bit of time to load because it must locate all interfaces that are defined in type libraries registered on your system. Note that you must use the Interface Selection wizard to implement an existing interface. If you type in the name of an existing interface, the wizard does not recognize this as an existing interface and assumes you are simply providing the object with a different interface name.
Description	Enter a description of the COM object you are creating.
Include Type Library	Check this box to generate a type library for this object. A type library contains type information that allows you to expose any object interface and its methods and properties to client applications.
Mark interface OleAutomation	Check this box to allow type library marshaling. This flag lets you avoid writing your own proxy-stub DLL for custom marshaling. When marking an interface as OleAutomation, you must ensure that it uses OLE Automation compatible types.

3.2.6.8 COM+ Event Interface Selection dialog box

On the COM+ Subscription Object wizard, use the Browse button to display the COM+ Event Interface Selection dialog box. It lists all event classes currently installed in the COM+ Catalog. The dialog also contains a Browse button that you can use to search for and select a type library containing the event interface.

3.2.6.9 COM+ Event Object Wizard

File ▶New▶Other...

The **COM+ Event Object** wizard creates a COM+ event object that can be called by a transactional server to generate events on clients. Because the project for a COM+ object can only contain other COM+ objects, you may be prompted to start a new project when you launch this wizard.

Item	Description
CoClassName	This is the name of your COM+ event object. Server objects that generate COM+ events create an instance of this object and call its events, which COM+ dispatches so that they fire on registered clients.
Interface	This is the name of the interface that defines the event handlers for all events managed by the COM+ event object. It is implemented by client event sinks, which means the wizard does not generate an implementation unit.
Description	Optionally, enter a brief description of your event objects so that clients can easily understand the purpose of the events.

When the **COM+ Event Object** wizard exits, you can define the methods of the generated interface using the **Type Library editor**. When defining this interface, the following rules must be followed:

- All method names must be unique across all interfaces of the event object.
- All methods must return an HRESULT value.
- The modifier for all method parameters must be blank.

3.2.6.10 COM+ Subscription Object Wizard

File▶New▶Other▶COM+ Subscription Object

You can create the COM+ event subscriber component using the COM+ Subscription Object wizard. You use this with a COM+ Event Object to receive notification of events fired by COM+ publisher applications.

Item	Description
Class Name	Enter the name of the class that will implement the event interface.
Threading Model	Choose the threading model. The threading model of a component determines how the methods of the component are assigned to threads to be executed.
Interface	In the Interface field, you can type the name of the event interface, or use the Browse button to bring up a list of all event classes currently installed in the COM+ Catalog. The COM+ Event Interface Selection dialog, which is displayed, also contains a Browse button that can be used to search for and select a type library containing the event interface.

Implement Existing Interface	When you select an interface, the wizard gives you the option of automatically implementing the interface supported by that event class. If you check the Implement Existing Interface checkbox, the wizard will automatically stub out each method in the interface for you.
Implement Ancestor Interfaces	When you select an interface, the wizard gives you the option of automatically implementing the ancestor interfaces of that event class. You can elect to have the wizard implement inherited interfaces by checking the Implement Ancestor Interfaces checkbox. Three ancestor interfaces are never implemented by the wizard: IUnknown, IDispatch, and IAppServer.
Description	Enter a brief description of your event subscriber component.

3.2.6.11 Customize New Menu

File ▶ New ▶ Customize

Use this dialog box to customize the content of the **File ▶ New** menu by dragging menu items from the center pane and dropping them on the right pane.

Item	Description
Gallery Items (left pane)	Displays the folders of gallery items that are available in the **Object Repository**. Click a folder to display its content in the center pane.
Menu Items (right pane)	Displays the items that are currently listed on the **File ▶ New** menu. To remove an item from the **File ▶ New** menu, drag it away from the list until its icon displays an X, and release the mouse button. To change the text for a menu item, double-click the text and enter new text. To add a separator bar between menu items, drag the **Separator** item from the center pane to the menu list.
Drag an item here	If you want to set a default application type, drag the item that represents the application type from the center pane and drop it on this button. To remove the default application, click the button.

3.2.6.12 Change Destination File Name

Use this dialog box to rename the selected file when it is copied to the destination directory. The rename will not occur until you copy the file to the destination directory. The source file will not be renamed.

If the file already exists in the destination directory with its original name, that file will remain in the destination directory until you delete it.

Item	Description
Source Filename	Indicates the name of the file in the source directory.
Subdirectory	Indicates the name of the subdirectory in which the source file resides.
Destination Filename	Enter the new name for the destination file.

See Also

Deploying ASP.NET Applications

Using the ASP.NET Deployment Manager

3

3.2.6.13 FTP Connection Options

Use this dialog box to specify FTP server connection information for the Deployment Manager.

Item	Description
Server	Specifies the internet host name or IP address for the FTP server. It will be prefixed with `FTP://`.
Port	Indicates the port to use on the FTP server for the connection. The default port is 21.
Passive Mode	Causes the connection to be established by your computer, rather than the FTP server. Check **Passive Mode** if your computer is protected by a firewall that would block a connection initiated by the FTP server.
Directory	Specifies the directory on the FTP server to which you want to copy files.
Anonymous Login	Logs into the FTP server with a user name of `anonymous` and a password of your email address, rather than using an actual account on the server. (The FTP server must be configured to accept anonymous logins.)
Email	If you checked **Anonymous Login**, specify your email address, for example, `YourName@domain.com`.
Username	Specifies the user name for the connection when not using an anonymous login.
Password	Specifies the password for the connection when not using an anonymous login.
Save Password (Clear Text)	Stores the password in the Deployment Manager file `(.bdsdeploy)`**as plain, unencrypted text in the project directory and may compromise security.**
Test	Tests the connection to the FTP server and provides feedback.

See Also

Deploying ASP.NET Applications

Using the ASP.NET Deployment Manager

3.2.6.14 Interface Selection Wizard

File ▶New▶Other...▶New COM Object

The **Interface Selection** wizard is accessed by clicking the List button on the **New COM Object** wizard. The **Interface Selection** wizard lets you select a predefined dual or custom interface that you want to implement with a COM object you are creating. The selected interface becomes the default interface of the newly created COM object. The COM object wizard adds skeletal method implementations for all the methods on this interface to the generated implementation unit. You can then fill in the bodies of these methods to provide an implementation of the interface.

Warning: The Interface Selection

wizard does not add the interface to your project’s type library. This means that when you deploy your object, you must also deploy the type library that defines your object’s interface.

Item	Description
Interface list	The wizard lists all the interfaces defined in registered type libraries. You can select any dual or custom interface from this list. Each interface is prefixed with the name of the file that contains its type library. This can be an executable (.exe), a library (.dll or .ocx), or a type library file (.tlb).

Add Library	If the desired interface is in a type library that is not currently registered, click the **Add Library** button, navigate to the type library that contains the interface you want, and click OK. This registers the selected type library and refreshes the interface list in the **Interface Selection** wizard.

3.2.6.15 New ASP.NET Application

File ▶New▶Other▶ASP NET Application

Use this dialog box to set options for new Web Forms applications.

Item	Description
Name	Specify the name of your ASP.NET application.
Location	Enter the root directory for your application.
Server	Indicate the default Web Server for new Web application.
View Server Options	Show/Hide server options such as access priviledges.
Alias	Specify the name used to gain access to the virtual directory.
Permissions	Specify access permissions for your application. These options are specific to Microsoft IIS server. To select options for the Cassini Web Server, use the drop-down list. **Read**: View directory of file content and properties. **Run scripts (such as ASP)**: Allows the Web Server to execute scripts. **Execute (such as ISAPI applications or CGI)**: Allows application execution for ISAPI or CGI type applications. **Write**: Allows you to upload files to the directory. **Browse**: Allows you to access list of files and directories.

3.2.6.16 New ASP.NET Content Page

File ▶New▶Other▶Delphi for .NET Projects▶ASP NET Content Page

Use this dialog box to create a new Content Page for an ASP.NET application.

Item	Description
Name	Specifies the name of the ASP.NET Content Page.
Master Page File	Specifies the Master Page referenced by the Content page.

3.2.6.17 New ASP.NET Generic Handler

File ▶New▶Other▶Delphi for .NET Projects▶ Generic Handler

Use this dialog box to create a new HTTP Generic Handler for an ASP.NET application.

Item	Description
Name	Specifies the name of the Generic Handler.

3.2.6.18 New ASP.NET Master Page

File ▶New▶Other▶Delphi for .NET Projects▶ASP NET Master Page

Use this dialog box to create a new Master Page for an ASP.NET application.

Item	Description
Name	Specifies the name of the ASP.NET Content Page.

3.2.6.19 New ASP.NET Master Page

File ▶New▶Other▶Delphi for .NET Projects▶ASP NET Page

Use this dialog box to create a new Page for an ASP.NET application.

Item	Description
Name	Specifies the name of the ASP.NET Page.

3.2.6.20 New ASP.NET User Control

File ▶New▶Other▶Delphi for .NET Projects▶ASP NET User Control

Use this dialog box to create a new User Control for an ASP.NET application.

Item	Description
Name	Specifies the name of the ASP.NET User Control.

3.2.6.21 New ASP.NET Web Service

File ▶New▶Other▶Delphi for .NET Projects▶ASP NET Web Service

Use this dialog box to create a new Web Service for an ASP.NET application.

Item	Description
Name	Specifies the name of the ASP.NET Web Service.

3.2.6.22 New Console Application

File ▶New▶Other

Use this dialog box to create an application that runs in a console window.

Item	Description
Source Type	Specifies the language to use for the main module of the application.

Use VCL	Creates an application that can contain VCL components. This option is available only if you choose **C++** as your **Source Type**. Checking this option causes the IDE to include `vcl.h` and to change the startup code and linker options for compatibility with VCL objects.
Multi Threaded	Specifies more than one thread of execution. This option is required if you check **Use VCL**.
Console Application	Creates a console window for your application.
Specify project source	Allows you to specify an existing source file for the console application. To specify a source file, check this option and click **[...]** to locate a file.

3.2.6.23 New DBWeb Control Wizard

File ▶New▶Other▶Delphi ASP Projects▶DBWeb Control Library

Use this dialog box to create a data aware WebControl. This DB Web Control can suppplement the DB Web Controls provided on the **Tool Palette**.

Item	Description
Control Name	The unit name to assign to the DB Web Control you are generating.
Bind to DataTable	Check this option to generate a DB Web Control that implements the IDBWebDataLink interface, which provides access to the DBDataSource and TableName properties for all controls.
Bind to DataColumn	Check this option to generate a DB Web Control that implements the IDBWebColumnLink interface, which provides access to the ColumnName property for controls that retrieve data from a specific column.
Support Lookup	Check this option to generate a DB Web Control that implements the IDBWebLookupColumnLink interface, which provides access to the LookupTableName, DataTextField, and DataValueField properties for lookup controls. This check box is only enabled when **Bind to DataColumn** is selected.

3.2.6.24 New Dynamic-link Library

File▶New▶Other

Use this dialog box to create a DLL project. A dynamic-link-library is a module of compiled code that provides functionality for applications.

Item	Description
Source Type	Specifies the language to use for the main module of the DLL.
Use VCL	Creates a DLL that can contain VCL components. This option is available only if you choose **C++** as your **Source Type**. Checking this option causes the IDE to include `vcl.h` and to change the startup code and linker options for compatibility with VCL objects.
Multi Threaded	Specifies more than one thread of execution. This option is required if you check **Use VCL**.
VC++ Style DLL	Sets the DLL entry point to `DLLMain`. Leave this option unchecked to use `DLLEntryPoint` as the entry point.

3.2.6.25 New Items

File ▶New▶Other

Use this dialog box to create a new project or other entity. The **New Items** dialog box displays project templates that are stored in the RAD Studio Object Repository.

Item	Description
Item Categories	Click a folder displayed in the **Item Categories** pane to display the types of entities that you can create.

Tip: Right-click the right pane to display a context menu for controlling the appearance of this dialog box.

See Also

New ASP.NET Application (see page 779)

New ASP.NET Content Page (see page 779)

New ASP.NET Generic Handler (see page 779)

New ASP.NET Master Page (see page 780)

New ASP.NET User Control (see page 780)

New ASP.NET Web Service (see page 780)

New Console Application (see page 780)

New DBWeb Control Wizard (see page 781)

New Dynamic-link Library (see page 781)

New Application (see page 782)

New Remote Data Module Object (see page 782)

New Thread Object (see page 783)

Active Form Wizard (see page 771)

Active Server Object Wizard (see page 772)

3.2.6.26 New Application

File ▶ New

Use this dialog box to specify a name and location for the new application.

Item	Description
Name	Enter the name for the application or accept the suggested application name.
Location	Enter a directory path or use the browse button to navigate to a directory. The default path for the application is displayed.

3.2.6.27 New Remote Data Module Object

File ▶ New ▶ Other

Use this dialog box to create a data module that can be accessed remotely as a dual-interface Automation server.

Item	Description
CoClass Name	Specifies the base name for the Automation interface and the remote data module. The class name for the remote data module will be the **CoClass Name** prepended with a T. The implementation class will inherit from an interface named using the **CoClass Name** prepended with an I. To enable a client application to access this interface, set the ServerName property of the client application's connection component to the **CoClass Name**.
Threading Model	Specifies how client calls are passed to the interface of the remote data module.
Model	Description
Single	The data module services one client request at a time. Because client requests are serialized by COM, you do not need to address thread conflicts.
Apartment	Each instance of the remote data module services one request at a time. The DLL might handle multple requests on separate threads if it creates multiple COM objects. Instance data is safe, but you must ensure that global memory is guarded against thread conflicts. This threading model is recommended if you use BDE-enabled datasets.
Free	The remote data module can receive simultaneous client requests on multiple threads. You must ensure that instance data and global memory are guarded against thread conflicts. This threading model is recommended if you use ADO datasets.
Both	This threading model is equivalent to the **Free** threading model, except all callbacks to client interfaces are serialized.
Neutral	Multiple clients can call the remote data module on different threads at the same time, but COM ensures that no two calls conflict. You must guard against thread conflicts involving global data and any instance data that is accessed by multiple interface methods. This threading model is only available under COM+. Otherwise, it is mapped to the **Apartment** threading model.
Description	Specifies the text that appears in the registry next to the `ProgID` for the application server interface. The **Description** text also acts as a help string for the interface in the type library.
Generate Event support code	Implements a separate interface for managing events.

See Also

Managing Events in Your Automation Object

3.2.6.28 New Thread Object

File ▶New▶Other▶Delphi Projects ▶Delphi Files▶Thread Object

Use this dialog box to define a thread class that encapsulates a single execution thread in a multi-threaded application.

Item	Description
Class Name	Type the full class name that you want to define. This dialog box does not prepend a T to the supplied class name; type the full class name, such as TMyThread, rather than typing MyThread.
Named Thread	If you want to name the thread, check **Named Thread** and then type a name in the **Thread Name** field. Naming the thread class adds a method to your thread class called SetName. When the thread starts running, it calls the SetName method first. Naming the thread class can make it easier to identify threads in the debugger **Thread Status** window.

Thread Name	Type the thread name you want to use.

Tip: Clicking OK

creates a new unit that defines a thread class with the name(s) supplied in the dialog. You must then supply the code that executes when the thread is run by writing the Execute method in the new unit.

3.2.6.29 Open

File ▶Open

Use this dialog box to locate and open a file. The title of this dialog box varies, depending on the function being performed.

Item	Description
Look in	Lists the current directory. Use the drop-down list to select a different drive or directory.
Go To Last Folder Visited	Moves to the last directory that you were in.
Up one directory	Moves up one directory level from the current directory.
Create New Folder	Creates a new subdirectory in the current directory.
View Menu	Displays a list of files and directories along with time stamp, size, and attribute information in one of five different ways: large icons, small icons, a vertical list, details (including time stamp, size, and attribute information), and thumbnails (a miniature version of a graphical image of a file).
Files	Displays the files in the current directory that match the wildcards in **File name** or the file type in **Files Of Type**. You can display a list of files (default) or you can show details for each file.
File name	Displays the name of the file you want to load. You can type wildcards to use as filters in the **Files** list box, or click the drop-down arrow to select a previously opened file.
Files of type	Displays the type of file you want to open. All files in the current directory of the selected type appear in the **Files** list box.
Open	Opens the selected file.

Tip: Press F1

in any list box or column to display tooltips with more information.

3.2.6.30 Package

File ▶New▶Other▶Delphi for .NET Projects▶Package

Use this dialog box to create a package.

Item	Description
Compile	Compiles the current package. If changes to the package are required, a dialog box appears that lists the changes that will be made to the package before it is compiled.
Add	Adds an item to the package.
Remove	Removes the selected item from the package.
Options	Displays the **Project Options** dialog box.

Install	Installs the current package as a design time package. If changes to the package are required, a dialog box appears that lists the changes that will be made to the package before it is compiled.
Contains	Displays the units included in the package. To add a unit to the package, click the **Add** button. To edit a unit's source code, double-click it.
Requires	Displays the other packages required by the current package. To add a package, click **Add**. To display a package in its own package editor, double-click it.

Tip: Right-click the package editor for a context menu with additional commands.

3.2.6.31 Print Selection

File ▶Print

Use this dialog box to print the current file.

Item	Description
Print selected block	Prints only the selected block of text in the current file.
Header/page number	Prints the file path and name as a heading on each page and numbers each page.
Line numbers	Prints line numbers on the printed pages.
Syntax print	Prints any syntax highlighting, such as bold keywords.
Use color	Prints syntax highlighting in color when printing to a color printer.
Wrap lines	Causes lines that exceed the width of the printed page to continue to the next line.
Left margin	Sets the left margin for the printed pages.
Setup	Opens the **Print Setup** dialog box.

3.2.6.32 Project Upgrade

File ▶Open

Use this dialog box to upgrade an older Delphi project that has no corresponding `.bdsproj` project file. When you upgrade the project, the `.bdsproj` file and other files and directories used by RAD Studio will be created in the project's directory.

3.2.6.33 Project Updated

File▶Open

This dialog box appears when a project from a previous version is automatically updated for RAD Studio. The following changes are made to the project:

- The encoding of project data is updated.
- References to `.lib`s, `.bpi`s, and `.csm`s are updated as needed.

  Note: For compatibility issues, see the release notes.

3.2.6.34 Remote Data Module Wizard

File ▶ New ▶ Other ▶ Remote Data Module

Use the Remote Data Module wizard to create a data module that can be accessed remotely as a dual-interface Automation server. A remote data module resides in the application server between a client and server in a multi-tiered database environment.

Item	Description
CoClass Name	Enter the base name for the Automation interface of your remote data module. The class name for your remote data module (a descendant of TRemoteDataModule) will be this name with a T prepended. It will implement an interface named using this base name with an I prepended. To enable a client application to access this interface, set the ServerName property of the client application's connection component to the base name you specify here.
Instancing	Use the instancing combo box to indicate how your remote data module application is launched. The following table lists the possible values:
Value	Meaning
Internal	The remote data module is created in an in-process server. Choose this option when creating a remote data module as part of an active library (DLL).
Single Instance	Only a single instance of the remote data module is created for each executable. Each client connection launches its own instance of the executable. The remote data module instance is therefore dedicated to a single client.
Multiple Instance	A single instance of the application (process) instantiates all remote data modules created for clients. Each remote data module is dedicated to a single client connection, but they all share the same process space.
Threading Model	Use the threading combo box to indicate how client calls are passed to your remote data module's interface. The following table lists the possible values:
Value	Meaning
Single	The data module only receives one client request at a time. Because all client requests are serialized by COM, you don't need to deal with threading issues.
Apartment	Each instance of your remote data module services one request at a time. However, the DLL may handle multiple requests on separate threads if it creates multiple COM objects. Instance data is safe, but you must guard against thread conflicts on global memory. This is the recommended model when using BDE-enabled datasets. (Note that when using BDE-enabled datasets you must add a session component with AutoSessionName set to Ttrue.)
Free	Your remote data module instances can receive simultaneous client requests on several threads. You must protect instance data as well as global memory against thread conflicts. This is the recommended model when using ADO datasets.
Both	The same as Free except that all callbacks to client interfaces are serialized.
Neutral	Multiple clients can call the remote data module on different threads at the same time, but COM ensures that no two calls conflict. You must guard against thread conflicts involving global data and any instance data that is accessed by multiple interface methods. This model is only available under COM+. Otherwise, it is mapped to the Apartment model.
Description	Enter the text that appears in the registry next to the ProgID for the application server interface. This text also acts as a help string for the interface in the type library.

Generate Event Support Code	Check this box to tell the wizard to implement a separate interface for managing events.

3.2.6.35 Satellite Assembly Wizard

File ▶ New ▶ Other ▶ Delphi for .NET Projects ▶ Satellite Assembly Wizard

Use this wizard to add one or more satellite assemblies to a project. Follow the instructions on each wizard page.

See Also

Using Translation Tools (see page 18)

Adding Languages to a Project (see page 169)

3.2.6.36 Revert to Previous Revision

This confirmation message appears when you click **Revert to previous revision** on the History tab of the Create Project dialog box.

Item	Description
Yes	Reverts the file or project to the previous saved version and abandons any changes you have made in the editor buffer. Note that reverting to the previous version does not automatically change the version in your repository. You still need to check the reverted version into your source repository.
No	Retains the contents of the editor buffer and cancels the revert operation.

3.2.6.37 Add New WebService

File ▶ New ▶ Other ▶ Delphi Projects ▶ Web Services ▶ SOAP Server Interface

Use this dialog box to define a new invokable interface and its implementation class. The dialog generates a new unit that declares an invokable interface and the implementation class. The interface descends from IInvokable, and the implementation class from TInvokableClass. It also generates the code to register the interface and implementation class with the invocation registry. After exiting the wizard, edit the generated interface and class definitions, adding in the properties and methods you want to expose as your Web Service.

Item	Description
Service name	Enter the name of the invokable interface (port type) that your Web Service application exposes to clients. This name is used as the name of the interface. It is also used to generate the name of the implementation class. Thus, for example, if you enter MyWebService, the wizard generates the definition of an invokable interface named MyWebService, and an implementation class named TMyWebServiceImpl.
Unit identifier	Enter the name of the unit that the wizard should create to contain the interface and implementation class definitions.
Generate comments	Optional. Adds comments to the unit generated by the wizard, indicating what the code does.
Generate sample methods	Optional. Adds sample code, as comments, to the unit generated by the wizard. You can then use the sample code as a guideline for defining and implementing the invokable interface and implementation class.

Service activation model	Select the activation model you want from the drop-down list: **Per Request** creates a new instance of your implementation class in response to each request it receives. That instance is freed after the request is handled. **Global Object** creates a single instance of your implementation class, which is used to handle all requests.

3.2.6.38 SOAP Data Module Wizard

File ▸New▸Other▸Delphi Projects▸Web Services▸SOAP Server Data Module

Use this dialog box to add a SOAP data module to a Web Service application. A SOAP data module allows a Web Service application to export database information as a Web Service. Client datasets on the client application can display and update this database information.

Item	Description
Module Name	Enter the base name of a TSoapDataModule descendant that your application creates. It is also the base name of the interface for that class. For example, if you specify the class name MyDataServer, the wizard creates a new unit declaring TMyDataServer, a descendant of TSoapDataModule, which implements IMyDataServer, a descendant of IAppServerSOAP.

3.2.6.39 New SOAP Server Application

File ▸New▸Other▸Delphi Projects▸Web Services▸SOAP Server Application

Use this dialog box to specify the type of server your Web Service application will work with.

Item	Description
ISAPI/NSAPI Dynamic Link Library	ISAPI and NSAPI Web server applications are DLLs that are loaded by the Web server. Client request information is passed to the DLL as a structure. Each request message is handled in a separate execution thread.
CGI stand-alone executable	A CGI stand-alone Web server application is a console application that receives client request information on standard input and passes the results back to the server on standard output. Each request message is handled by a separate instance of the application.
Web App Debugger executable	The Web Application Debugger provides an easy way to monitor HTTP requests, responses, and response times. The Web Application Debugger takes the place of the Web server. Once you have debugged your application, you can convert it to one of the other types of Web application and install it with a commercial Web server.
Class Name	If selecting **Web App Debugger executable**, provide a class name for the debugger to use to call your Web module.

3.2.6.40 Save As

File ▸Save As

Use this dialog box to to save the active file.

Item	Description
Save in	Lists the current directory. Use the drop-down list to select a different drive or directory.

Go To Last Folder Visited	Moves to the last directory that you were in.
Up one directory	Moves up one directory level from the current directory.
Create New Folder	Creates a new subdirectory in the current directory.
View Menu	Displays a list of files and directories along with time stamp, size, and attribute information in one of five different ways: large icons, small icons, a vertical list, details (including time stamp, size, and attribute information), and thumbnails (a miniature version of a graphical image of a file).
Files	Displays the files in the current directory that match the wildcards in **File name** or the file type in **Files Of Type**. You can display a list of files (default) or you can show details for each file.
File name	Displays the name of the file you want to save. You can type wildcards to use as filters in the **Files** list box, or click the drop-down arrow to select a previously saved file.
Save as type	Displays the type of file you want to save. All files in the current directory of the selected type appear in the **Files** list box.
Save	Saves the selected file.

Tip: Press F1

in any list box or column to display tooltips with more information.

3.2.6.41 Select Directory

Use this dialog box to to choose a working directory for your new project.

Item	Description
Directory Name	Displays the current directory. If you enter a directory that does not exist, the product creates it.
Directories	Lists the current directory.
Files (*.*)	Lists all the files in the current directory. You cannot select any of these files. The product displays this file list so you know the contents of the current directory.
Drives	Lists all the available drives. You can select one of the available drives.

3.2.6.42 Transactional Object Wizard

File ▶New▶Other...

Use the New Transactional Object wizard to create a server object that runs under MTS or COM+. Transactional objects are used in distributed applications to make use of the special services supplied by MTS or COM+ for resource management, transaction support, or security.

Item	Description
CoClassName	Specify the name for the object that you want to implement. The wizard generates an interface that has this name with an ‘I’ prepended and an implementation class that has this name with a ‘T’ prepended.
Threading Model	Choose the threading model to indicate how MTS or COM+ serializes calls to the transactional object’s interface. The threading model you choose determines how the object is registered. You must make sure that your object implementation adheres to the model selected. Threading model values are shown below.

Transaction model	Specify the transaction attribute that is assigned to your object when you register it. The possible values are shown below.
Generate event support code	Do not describe standard buttons, such as **OK** and **Cancel**.

The Threading Model combo box can take the following values:

Model	Description
Single	Your code has no thread support. Only one client thread can be serviced at a time.
Apartment	Under COM+, each object instantiated by a client is accessed by one thread at a time. You must protect against multiple threads accessing global memory, but objects can safely access their own instance data (object properties and members). Under MTS, it is also the case that all client calls use the thread under which the object was created.
Both	The same as Apartment except that callbacks to clients are serialized as well.
Neutral	Multiple clients can call the object on different threads at the same time, but COM ensures that no two calls conflict. You must guard against thread conflicts involving global data and any instance data that is accessed by more than one method. This model should not be used with objects that have a user interface. This model is only available under COM+. Under COM, it is mapped to the Apartment model.

Note: The serialization of calls to your object is also influenced by how it participates in activities. Under MTS, objects are always synchronized by the current activity. Under COM+, this can be configured using the COM+ page of the Type Library Editor

or the COM+ Component Manager. The Transaction model combo box can take the following values:

Value	Meaning
Requires a transaction	The object must execute within the scope of a transaction. When a new object is created, its object context inherits the transaction from the context of the client. If the client does not have a transaction context, a new transaction context is automatically generated.
Requires a new transaction	The object must execute within its own transaction. When a new object is created, a new transaction context is automatically created as well, regardless of whether its client has a transaction. The object never runs inside the scope of its client's transaction. Instead, the system always creates independent transactions for the new objects.
Supports Transactions	The object can execute within the scope of its client's transactions. When a new object is created, its object context inherits the transaction from the context of the client if there is one. Otherwise, the object is not created in the scope of a transaction.
Does not support transactions	Under MTS, this setting behaves like Transactions Ignored under COM+ (see above). Under COM+, the object can't run in the context of a transaction at all. If the client has a transaction, attempts to create the object will fail.
Ignores Transactions	The object does not run within the scope of transactions. When a new object is created, its object context is created without a transaction, regardless of whether the client has a transaction. This model is not supported under MTS.

3.2.6.43 Use Unit

File ▶Use Unit

Use this dialog box to make the contents of a unit available to the current unit. This dialog lists of all units in the project that are not currently used or included by the current unit. You can only use units that are part of the current project.

3.2.6.44 WSDL Import Options

Use this dialog box to specify information that the importer needs to connect to the server that publishes a WSDL document or to configure the way the wizard generates code to represent the definitions in a WSDL document.

Connection Tab

Use the **Connection** tab page to provide information the wizard needs for connecting to the server that hosts the WSDL document.

Item	Description
User Name	Specify the user name to use if the WSDL document is on a secure server that requires authentication.
Password	Specify the password to use with User Name when the WSDL document is on a secure server that requires authentication.
Proxy	Specify the host names for any proxy servers that must forward requests to the URL specified on the Source page of the Web Services Import dialog.

Code Generation Tab

Use the **Code Generation** tab page to configure how the importer translates between the definitions in the WSDL document and the native code that it generates.

Item	Description
Declare Namespace	This option is only valid when the importer generates C++ code. When checked, it puts all generated type definitions in a C++ namespace that is named after the imported service.
One OutParam is Return	When checked, the importer maps operations with a single output message into functions where the output message is the return value. If this is not checked, the output message is mapped to an output parameter.
Unwind Literal Params	In document literal encoding, the Web Service does not describe operations. Rather, it describes two records, one that describes the expected input and one that describes the output. When Unwind Literal Params is checked, the importer converts these two records into method calls.
Generate Destructors	When checked, the importer generates destructors on the classes that represent types. These destructors free any nested members whose types are classes or arrays of classes. The generated destructors simplify the work you must do when freeing instances of classes that represent types, because you do not need to explicitly free class members that also use classes to represent the remotable type.
Ignore Schema Errors	When checked, the importer attempts to import WSDL documents that are not well-formed. Often the importer can deduce the appropriate information even when the schema is badly formed.
Warnings Comments	When checked, the importer adds warning messages to the comments it puts in the top of generated files. These warnings describe problems such as invalid type definitions in the WSDL document when Ignore Schema Errors is checked, problems encountered when unwinding literal parameters when Unwind Literal Params is checked, and so on.
Emit Literal Types	In document literal encoding, the Web Service does not describe operations. Rather, it describes two records, one that describes the expected input and one that describes the output. When Generate Literal Types is checked, the importer generates type definitions for these two records, even if it converts them to method calls (that is, even if Unwind Literal Params is checked).

Ambiguous Types as Array	In some cases when the WSDL document does not make consistent use of schema definitions, the importer has problems importing array types, which results in a type that is represented by a class with no members. When Ambiguous Types as Array is checked, the importer compensates by changing these empty classes to arrays. This option is always safe to use unless the WSDL document describes a Web Service that uses document literal encoding. Document literal encoding can also give rise to an empty class that represents a procedure. If you check Ambiguous Types as Array when importing a WSDL document for a Web Service that uses document literal encoding, the resulting generated code may not work.
Generate Server Implementation	When checked, the importer generates implementation classes for the imported interfaces. Use this option when writing a server that implements a Web Service that is already defined in a WSDL document.
Map String to WideString	When checked, the importer maps all string types to WideString values. When Unchecked, the importer uses the string type instead. WideString values may be required to handle values that use extended characters. If string values do not use extended characters, it is more efficient to use the string type.

3.2.6.45 WSDL Import Wizard

File ▶ New ▶ Other ▶ Delphi Projects ▶ Web Services ▶ WSDL Importer

Use this wizard to import a WSDL document or XML schema that describes a Web Service. Once you have imported the WSDL document or XML schema, the wizard generates all the interface and class definitions you need for calling on those Web Services using a remote interfaced object (THTTPRIO). You can also tell the wizard to generate skeletal code you can complete to create a Web Service application (for example, if you want to implement a Web Service that is already defined in a WSDL document).

Source Page

The **Source** page of the wizard lets you specify the name of the WSDL document or XML schema to import.

Item	Description
Location of WSDL File or URL	Enter either a WSDL file name or the URL where the document is published. Click the ellipsis button next to the edit box to browse for a file location. If you do not know the URL of the WSDL document, or (for client applications) if you want to include fail-over support, click the **Search UDDI** button to launch the UDDI browser. When you import a WSDL document using the UDDI browser, it initializes the location in the WSDL importer, and causes the importer to generate code that stores the location of the UDDI entry where you found the document. After entering a file name, you can click the **Next** button to move to the **Preview** page or click the **Options** button to provide information the wizard needs for connecting to a server that contains the WSDL document or for configuring how the wizard generates code to represent the definitions in that document.

Preview Page

The **Preview** page of the wizard lets you preview the code it generates for the definitions in the specified WSDL document. It lists only those definitions for which it knows how to generate code. When you are finished viewing the generated code, you can move back to the **Source** page to select a different WSDL document, click the **Options** button to change the connection information or configure how the wizard generates code for the definitions in the WSDL document, or click the **Finish** button, which tells the wizard to define and register invokable interfaces and native type definitions.

Item	Description
WSDL Components	Select the item for which you want to preview the generated code. You can select a service to see all the definitions generated for that service, or you can select a single item that is a part of that service (for example, a single interface, method, or type definition).
Code Preview	This tab page shows the code that the wizard generates for the selected item.
Attributes	This tab page shows information about the selected item such as its name, the namespace in which it is defined, and details about its definition, such as the name of the class or type the wizard uses to represent a defined type, the parameters and Soap Action for a method (operation), binding information for an interface (port type), and so on.
Options	Display the **WSDL Import Options** dialog, where you can specify the information the importer needs to connect to the server that publishes the WSDL document or configure the way the wizard generates code.

3.2.6.46 New Web Server Application

File ▶New▶Other▶Delphi Projects▶New▶Web Server Application

Use this dialog box to specify the type of server your Web server application will work with.

Item	Description
ISAPI/NSAPI Dynamic Link Library	ISAPI and NSAPI Web server applications are shared objects that are loaded by the Web server. Client request information is passed to the DLL as a structure and evaluated by TISAPIApplication. Each request message is handled in a separate execution thread. Selecting this type of application adds the library header of the project files and required entries to the uses list and exports clause of the project file.
CGI Stand-alone executable	A CGI stand-alone Web server application is a console application that receives client request information on standard input and passes the results back to the server on standard output. This data is evaluated by TCGIApplication. Each request message is handled by a separate instance of the application. In Delphi, selecting this type of application adds the required entries to the uses clause of the project file and adds the appropriate $APPTYPE directive to the source.
Web App Debugger executable	The Web Application Debugger provides an easy way to monitor HTTP requests, responses, and response times. The Web Application Debugger takes the place of the Web server. Once you have debugged your application, you can convert it to one of the other types of Web application and install it with a commercial Web server. When you select this type of application, you must specify a Class Name for the debugger executable. This is simply a name used by the Web App Debugger to refer to your application. Most developers use the application's name as the Class Name.

3.2.6.47 Add New WebService

File ▶New▶Other▶Delphi Projects▶WebServices▶SOAP Server Interface

Use this dialog box to generate a new unit that declares a invokable interface and its implementation class. The interface descends from IInvokable, and the implementation class from TInvokableClass. It also generates the code to register the interface and implementation class with the invocation registry. After exiting the wizard, edit the generated interface and class definitions, adding in the properties and methods you want to expose as your Web Service.

Item	Description
Service name	Enter the name of the invokable interface (port type) that your Web Service application exposes to clients. This name is used as the name of the interface. It is also used to generate the name of the implementation class. For example, if you enter MyWebService, the wizard generates the definition of an invokable interface named MyWebService, and an implementation class named TMyWebServiceImpl.
Unit identifier	Enter the name of the unit that the wizard should create to contain the interface and implementation class definitions.
Generate comments	Adds comments to the unit that it generates telling you what the generated code does.
Generate sample methods	Adds comments to the unit that show sample code similar to the code you should add to define and implement your invokable interface and implementation class.
Service activation model	Select the activation model you want from the drop-down list: **Per Request** creates a new instance of your implementation class in response to each request it receives. That instance is freed after the request is handled. **Global Object** creates a single instance of your implementation class, which is used to handle all requests.

3.2.6.48 Application Module Page Options/New WebSnap Page Module

Use this dialog box to define the basic properties of a page module. The dialog title varies based on how you accessed the dialog.

Item	Description
Type	The producer type for the page can be set to one of AdapterPageProducer, DataSetPageProducer, InetXPageProducer, PageProducer, or XSLPageProducer.
Script Engine	If the selected page producer supports scripting, use the Script Engine drop-down list to select the language used to script the page. AdapterPageProducer supports only JScript.
New File	Creates a template file and manages it as part of the unit. A managed template file will appear in the project manager and have the same file name and location as the unit source file. Uncheck **New File** if you want to use the properties of the producer component (typically, the HTMLDoc or HTMLFile property).
Template	When **New File** is checked, choose the default content for the template file from the **Template** drop-down. The Standard template displays the title of the application, the title of the page, and hyperlinks to published pages.
Name	Enter a page name and title for the page module. The page name is used to reference the page in an HTTP request or within the application's logic.
Title	Specifies the name that the end user will see when the page is displayed in a browser.
Published	Check Published to allow the page to automatically respond to HTTP requests where the page name matches the pathinfo in the request message.
Login Required	Requires the user to log on before the page can be accessed.
Creation	Displayed only on the **New WebSnap Page Module**. This parameter controls when an instance of this module is created. If you want the instance created only when it is referenced, select On Demand. If you want the instance created on startup, select Always.

Caching	Displayed only on the **New WebSnap Page Module**. This parameter controls when a module is destroyed. At runtime, the instance of this module can be either kept in cache, or removed from memory when the request has been serviced. Select Cache Instance to keep the instance in memory even if there are no current references to it. Select Destroy Instance to allow the server to remove the instance from memory if there are no references to it.

See Also

WebSnap Overview

Building a WebSnap Application

3.2.6.49 New WebSnap Application

File ▶ New ▶ Other ▶ Delphi Projects ▶ WebSnap ▶ WebSnap Application

Use this dialog box to configure a new WebSnap application.

Item	Description
ISAPI/NSAPI Dynamic Link Library	ISAPI and NSAPI Web server applications are shared objects that are loaded by the Web server. Client request information is passed to the DLL as a structure and evaluated by TISAPIApplication. Each request message is handled in a separate execution thread. Selecting this type of application adds the library header of the project files and required entries to the uses list and exports clause of the project file.
CGI Stand-alone executable	A CGI stand-alone Web server application is a console application that receives client request information on standard input and passes the results back to the server on standard output. This data is evaluated by TCGIApplication. Each request message is handled by a separate instance of the application. In Delphi, selecting this type of application adds the required entries to the uses clause of the project file and adds the appropriate $APPTYPE directive to the source.
Web App Debugger executable	The **Web Application Debugger** provides an easy way to monitor HTTP requests, responses, and response times. The **Web Application Debugger** takes the place of the Web server. Once you have debugged your application, you can convert it to one of the other types of Web application and install it with a commercial Web server. When you select this type of application, you must specify a class name for the debugger executable. This is simply a name used by the **Web Application Debugger** to refer to your application. Most developers use the application name as the class name.
Page Module	Displays a Web data module that includes a PageProducer, WebAppServices, ApplicationAdapter, LogicalPageDispatcher, and AdapterDispatcher component. With a Web Page Module, on the **Code Editor** you can view a Web page's unit, HTML code, and a preview the Web page after the module has been compiled and run.
Data Module	Displays a Web data module that includes a PageProducer, WebAppServices, ApplicationAdapter, LogicalPageDispatcher, and AdapterDispatcher component.
Components	Displays the **Web App Components** dialog box so you can select one or more components to add functionality to your application.
Page Name	If the selected application module type is page module, you can associate a name with the page by entering a name in the this field.
Page Options	Displays the **Application Page Module Options** dialog box, allowing you to define the basic properties of a page module.
Caching	At runtime, the instance of this module can be either kept in cache, or removed from memory when the request has been serviced. Select one of the following: **Cache Instance** stores the module in memory between user sessions. **Destroy Instance** removes the module from memory as soon as the active session ends.

See Also

WebSnap Overview

Building a WebSnap Application

3.2.6.50 New WebSnap Data Module

Use this dialog box to specify how the server handles the creation and destruction of your data module.

Item	Description
Creation	Controls when an instance of this module is created: **On Demand** creates the instance only when it is referenced. **Always** creates the instance on startup.
Caching	Controls when a module is destroyed. At runtime, the instance of this module can be either kept in cache, or removed from memory when the request has been serviced. **Cache Instance** keeps the instance in memory even if there are no current references to it. **Destroy Instance** allows the server to remove the instance from memory if there are no references to it.

See Also

WebSnap Overview

Building a WebSnap Application

3.2.6.51 Web App Components

Use this dialog box to select component categories and to select specific components in each of the following categories (some categories offer only one choice).

Item	Description
Application Adapter	Contains the field and action components that are available through the Application script variable.
End User Adapter	Provides information about a user such as their name, access rights, and whether they are logged in. TEndUserAdapter calls event handlers to retrieve user information.
Page Dispatcher	Dispatches HTTP requests that reference a Web page module by name.
Adapter Dispatcher	Handles HTML form submissions, and requests for dynamic images, by calling adapter action and field components.
Dispatch Actions	Passes an HTTP request message to the appropriate action items that assemble a response.
Locate File Service	Controls the location of templates and include files at runtime.
Sessions Service	Stores information about end user data that is needed for a short period of time. For example, the TSessionsService can be used to keep track of all users that are currently logged in and automatically log a user out after a period of inactivity..
User List Service	Contains a list of user names, password, and access rights. It is used to validate login and check access rights for a particular user.

See Also

WebSnap Overview

Building a WebSnap Application

3.2.6.52 XML Data Binding Wizard Options

File ▶New▶Other▶Delphi Project▶New▶XML Data Binding▶Options button

Use this dialog box to determine how the **XML Data Binding Wizard** generates interfaces and implementation classes to represent an XML document or schema.

Item	Description
Category	Choose a category of options such as **Code Generation**. The table at the right displays the options for the selected category.
Options table	Edit the values in the second column of the Options table to change one of the options that the wizard uses. When the **Category** is **Data Type Map**, this table displays the types that the wizard generates for each XML type that appears in the XML schema. You can edit these values to change the mapped type. For example, you may want to change a data type to Variant so that you can distinguish between an empty string and a blank value.
Property Get Prefix	Controls the name the wizard assigns to the methods it creates for reading property values. These methods consist of the **Get** prefix followed by the name of the property (element).
Property Set Prefix	Controls the name the wizard assigns to the methods it creates for writing property values. These methods consist of the **Set** prefix followed by the name of the property (element).
Class Name Prefix	Controls the names that the wizard assigns to implementation classes for nodes. These classes are given the name of the element or attribute with the **Class Name Prefix** prepended.
Interface Prefix	Controls the names that the wizard assigns to interfaces. These interfaces are given the name of the element with **Interface Prefix** prepended.
Node List Type Suffix	Controls the names that the wizard assigns to the classes and interfaces it generates for repeating collections of child nodes. These classes get the name of the child node tag, with **Node List Type Suffix** appended (and **Class Name Prefix** or **Interface Prefix** prepended).
Node Interface Base	Specifies the interface that is used as a base from which all generated interfaces for nodes are derived.
Node Class Base	Specifies the class that is used as a base class from which all generated implementation classes are derived. **Node Class Base** should implement the interface specified by **Node Interface Base**.
Collection Intf. Base	Specifies the interface that is used as a base from which all generated interfaces that represent repeating child nodes are derived.
Collection Class Base	Specifies the class that is used as a base class from which all classes that represent repeating child nodes
Default Data Type	Specifies the type that is assigned to nodes by default on the second page of the wizard.

3.2.6.53 XML Data Binding Wizard, page 1

File ▶New▶Other▶Delphi Project▶New▶XML Data Binding

Use this wizard to generate interface and class definitions that correspond to the structure of an XML document or schema. The wizard generates a global function that returns the interface for the root element of the document.

After you use the wizard to create these definitions, you can use the classes and interfaces to work with XML documents that have the structure of the specified document or schema.

Item	Description
Schema or XML Data File	Enter the file name of a schema or XML document for which you want the wizard to generate interfaces and implementation classes. Beside the edit control is a browse button (labeled with ellipsis) that you can click to browse for an XML document or schema file.
Use XDB Settings File	Indicates whether the wizard should be initialized to reflect the choices you made the last time you used the wizard and saved your settings. When checked, the wizard starts by using the last XDB file that you saved using the third page of the wizard.
Options	Displays the **XML Data Binding Wizard Options** dialog box. You can select various options that influence how the wizard generates code for the interfaces and implementation classes in your XML document or schema.

3.2.6.54 XML Data Binding Wizard, page 2

File ▶New▶Other▶Delphi Project▶New▶XML Data Binding

Use this wizard page to specify what code the wizard generates.

Item	Description
Schema components	Displays a hierarchy of elements for which the wizard can generate interfaces and classes. This hierarchy is divided into complex elements (nodes that correspond to tags that have child nodes) and simple elements (simple data types that the schema defines for elements in the XML document). Nodes for complex types can be expanded to display nodes for the child elements. When you select a node in the Schema Components hierarchy, the right side of the dialog displays detailed information about the node and lets you indicate what code, if any, the wizard should generate for that node.
Source Name	Displays the name of the type or tag in the XML schema. Edit the value if you want the wizard to create or modify the schema file.
Source Datatype	Displays the type of the selected node, as defined in the XML schema. Edit the value if you want the wizard to create or modify the schema file.
Documentation	Displays any comments from the XML schema to describe the type or node. Edit the value if you want the wizard to create or modify the schema file.
Generate Binding	Creates an interface and implementation class for a selected complex type, or a property on the parent interface and class for simple elements that are children of a complex type.
Identifier Name	Specifies the name of the interface to generate for a top-level complex type. For the children of a complex type, **Identifier Name** specifies the name of the property created for this child in the parent element's interface.
Doc Element Type	This option is available only for top-level complex types and indicates the type of the document element (the root of the data hierarchy).
Element Name	Specifies the tag name of the document element.
Data Type	For any child node, indicates the type for the property that represents this child element. If the child represents an element node that has children of its own, the drop-down list lets you select any interface type that the wizard generates for a complex type. If the child represents a simple element, the drop-down list lets you select a type such as Integer, String, or Variant. Note that representing simple child elements as Variants allows your application to distinguish between elements with an empty string for a value, and elements that do not appear in a particular document (Null Variants).
Repeating	For child elements that represent complex types, indicates whether the parent node can have more than one child node of this type.
Access Mode	For child nodes that represent simple elements (as opposed to complex types), indicates whether the generated property is read/write or read-only.

Native Type	When a simple type is selected, specifies the data type that the wizard uses to represent values of that type.
Options	Displays the **XML Data Binding Wizard Options** dialog box. You can select various options that influence how the wizard generates code for the interfaces and implementation classes in your XML document or schema.

3.2.6.55 XML Data Binding Wizard, page 3

File ▶New▶Other▶Delphi Project▶New▶XML Data Binding

Use this wizard page to confirm the choices you have made, specify unit-wide code generation options, indicate where you want your choices saved, and tell the wizard to generate code to represent the XML document or schema.

Item	Description
Generated Interfaces	indicates what interfaces the wizard will generate. When you select an interface in this control, you can see the interface definition the wizard will generate in the **Code Preview** control.
Code Preview	Displays the code that the wizard will generate for the currently selected interface in the **Generated Interfaces** control.
Do not store settings	Generates code for the choices you have made with the wizard, but does not save the choices.
Store in XML schema	Updates the schema file with information about the choices you have made.
Store in file	Enter the name of a schema file where the wizard stores information about your choices. This schema file is independent of the XML document or schema file you selected on the first page of the wizard. The wizard uses this file to initialize itself the next time you use it.
Options	Displays the **XML Data Binding Wizard Options** dialog box. You can select various options that influence how the wizard generates code for the interfaces and implementation classes in your XML document or schema.
Finish	Exits the wizard and generates the interfaces and implementation classes for your XML document or schema.

3.2.6.56 Close

File ▶Close

File▶Close All

Closes the current open project or all the open projects.

Item	Description
Close	Closes the active project.
Close All	Closes all open projects.

3.2.6.57 Exit

File▶Exit

Closes the IDE and all open projects and files.

3.2.6.58 New

File ▶New

Creates a project of the selected type.

Item	Description
C++ Builder Projects Only	
Package - C++ Builder	Creates a skeleton of a C++ package project.
Unit - C++Builder	Creates a new C++ unit skeleton. Typically you create a new unit after creating a new package or application.
VCL Forms Application - C++Builder	Creates a new C++ form-based application using VCL components.
Form - C++Builder	Creates a new C++ VCL form. Typically you would create a new form after creating a new package or application.
Delphi for Win32 Projects Only	
Package - Delphi for Win32	Creates a skeleton for a Delphi package project.
Unit - Delphi for Win32	Creates a new Delphi unit skeleton (using Pascal source). Typically you would create a new unit after creating a new package or application.
VCL Forms Application - Delphi for Win32	Creates a new form-based application using VCL components.
Form - Delphi for Win32	Creates a new Delphi VCL form. Typically you create a new form after creating a new package or application.
Delphi for .NET Projects Only	
ASP.NET Web Application - Delphi .NET	Creates a new web form application using ASP.NET. A Web Form Application runs in a browser.
VCL Forms Application - Delphi for .NET	Creates a new form-based application using VCL components.
All Personalities	
Other	The **New Items** dialog box, which lists every file or project the IDE understands. Examples: XML files, text files, and C++ projects.
Customize	Opens the **Customize New Menu** dialog box that allows you to customize how projects and files are listed in the **File ▶New** menu.

See Also

Packages (see page 640)

Program and Units (see page 683)

VCL Overview

Customize New Menu (see page 777)

3.2.7 HTML Elements

Topics

Name	Description
A (Anchor) HTML Element (↗ see page 801)	The A (anchor) element designates the start or destination of a hypertext link. The anchor element requires the HREF= or the NAME= attribute to be specified and provides the following attributes and events.
Unit (↗ see page 803)	Use this dialog box to specify a value and unit of measure for the CSS attribute selected from the **Code Completion** window.
DIV HTML Element (↗ see page 803)	The DIV element is used with the CLASS= attribute to represent different kinds of containers and provides the following attributes and events.
HR HTML Element (↗ see page 804)	The HR element draws a horizontal rule in a document and provides the following attributes and events.
IMG HTML Element (↗ see page 805)	The IMG element embeds an image or a video clip in the document and provides the following attributes and events.
INPUT HTML Element (↗ see page 806)	The INPUT element specifies a form input control and provides the following attributes and events.
SELECT HTML Element (↗ see page 808)	The SELECT element specifies a list box or dropdown list and provides the following attributes and events.
SPAN HTML Element (↗ see page 809)	The SPAN element lets you to define your own method of rendering, using style sheets and provides the following attributes and events.
TABLE HTML Element (↗ see page 810)	The TABLE element specifies that the contained content is organized into a table with rows and columns. Use the TR, TD, and TH elements in the container to create the rows, columns, and cells. The TABLE element provides the following attributes and events. **Tip:** From the IDE main menu, choose View->Toolbars->HTML Table to display a toolbar for formatting tables.
TEXTAREA HTML Element (↗ see page 811)	The TEXTAREA element specifies a multi-line text input control and provides the following attributes and events.

3.2.7.1 A (Anchor) HTML Element

The A (anchor) element designates the start or destination of a hypertext link. The anchor element requires the HREF= or the NAME= attribute to be specified and provides the following attributes and events.

Item	Description
accesskey	Assigns an access key to an element. An access key is a single character from the document character set. Authors should consider the input method of the expected reader when specifying an accesskey.
charset	Specifies the character encoding of the resource designated by the link.
class	Assigns a class name or set of class names to an element. Any number of elements may be assigned the same class name or names. Multiple class names must be separated by white space characters.
coords	Specifies the position and shape on the screen relative to the top, left corner of the object. The number and order of values depends on the shape being defined. Possible combinations: **rect** left-x, top-y, right-x, bottom-y. **circle** center-x, center-y, radius. When the radius value is a percentage value, user agents should calculate the final radius value based on the associated object's width and height. The radius should be the smaller value of the two. **poly** x1, y1, x2, y2, ..., xN, yN. The first x and y coordinate pair and the last should be the same to close the polygon. When these coordinate values are not the same, user agents should infer an additional coordinate pair to close the polygon.

dir	Specifies the direction of directionally neutral text in an element's content and attribute values: **ltr** indicates left-to-right text or table. **rtl** indicates right-to-left text or table.
href	Specifies the location of a Web resource and definines a link between the current element (the source anchor) and the destination anchor defined by this attribute.
hreflang	Specifies the base language of the resource designated by href and may only be used when href is specified.
id	Assigns a name to an element. This name must be unique in a document.
lang	Specifies the base language of an element's attribute values and text content. The default value of this attribute is unknown.
name	Assigns the control name.
onblur	The onblur event occurs when an element loses focus either by the pointing device or by tabbing navigation.
onclick	The onclick event occurs when the pointing device button is clicked over an element.
ondblclick	The ondblclick event occurs when the pointing device button is double clicked over an element.
onfocus	The onfocus event occurs when an element receives focus either by the pointing device or by tabbing navigation.
onkeydown	The onkeydown event occurs when a key is pressed down over an element.
onkeypress	The onkeypress event occurs when a key is pressed and released over an element.
onkeyup	The onkeyup event occurs when a key is released over an element. This attribute may be used with most elements.
onmousedown	The onmousedown event occurs when the pointing device button is pressed over an element.
onmousemove	The onmousemove event occurs when the pointing device is moved while it is over an element.
onmouseout	The onmouseout event occurs when the pointing device is moved away from an element.
onmouseover	The onmousemove event occurs when the pointing device is moved while it is over an element.
onmouseup	The onmouseup event occurs when the pointing device button is released over an element.
rel	Specifies the relationship from the current document to the anchor specified by the **href** attribute. The value of this attribute is a space-separated list of link types.
rev	Specifies a reverse link from the anchor specified by the **href** attribute to the current document. The value of this attribute is a space-separated list of link types.
shape	When the **type** attribute is set to **image**, this attribute specifies the location of the image to be used to decorate the graphical submit button.
style	Specifies style information for the current element. The syntax of the value of the style attribute is determined by the default style sheet language.
tabindex	Specifies the position of the current element in the tabbing order for the current document. This value must be a number between 0 and 32767. User agents should ignore leading zeros.
title	Specifies advisory information about the element for which it is set.
type	Provides an advisory hint as to the content type of the content available at the link target address. It allows user agents to opt to use a fallback mechanism rather than fetch the content if they are advised that they will get content in a content type they do not support. Authors who use this attribute take responsibility to manage the risk that it may become inconsistent with the content available at the link target address. For the current list of registered content types, please consult the www.w3.org web site.

See Also

W3C HTML Home Page

3.2.7.2 Unit

Use this dialog box to specify a value and unit of measure for the CSS attribute selected from the **Code Completion** window.

Item	Description
Value	Select a number of units for the attribute.
Units	Select one of the following units of measure for the attribute: **em** equal to the font size of the current element **ex** approximately half the height of the font-size **px** pixel **in** inch **cm** centimeter **mm** millimeter **pt** point (one point is 1/72 of an inch) **pc** pica (about 12 points)

3.2.7.3 DIV HTML Element

The DIV element is used with the CLASS= attribute to represent different kinds of containers and provides the following attributes and events.

Item	Description
align	Specifies the horizontal alignment of its element with respect to the surrounding context. Possible values are **left**, **right**, **center**, and **justify**.
class	Assigns a class name or set of class names to an element. Any number of elements may be assigned the same class name or names. Multiple class names must be separated by white space characters.
dir	Specifies the direction of directionally neutral text in an element's content and attribute values: **ltr** indicates left-to-right text or table. **rtl** indicates right-to-left text or table.
id	Assigns a name to an element. This name must be unique in a document.
lang	Specifies the base language of an element's attribute values and text content. The default value of this attribute is unknown.
onclick	The onclick event occurs when the pointing device button is clicked over an element.
ondblclick	The ondblclick event occurs when the pointing device button is double clicked over an element.
onkeydown	The onkeydown event occurs when a key is pressed down over an element.
onkeypress	The onkeypress event occurs when a key is pressed and released over an element.
onkeyup	The onkeyup event occurs when a key is released over an element. This attribute may be used with most elements.
onmousedown	The onmousedown event occurs when the pointing device button is pressed over an element.

onmousemove	The onmousemove event occurs when the pointing device is moved while it is over an element.
onmouseout	The onmouseout event occurs when the pointing device is moved away from an element.
onmouseover	The onmousemove event occurs when the pointing device is moved while it is over an element.
onmouseup	The onmouseup event occurs when the pointing device button is released over an element.
style	Specifies style information for the current element. The syntax of the value of the style attribute is determined by the default style sheet language.
title	Specifies advisory information about the element for which it is set.

See Also

W3C HTML Home Page

3.2.7.4 HR HTML Element

The HR element draws a horizontal rule in a document and provides the following attributes and events.

Item	Description
align	Specifies the horizontal alignment of its element with respect to the surrounding context. Possible values are **left**, **right**, **center**, and **justify**.
class	Assigns a class name or set of class names to an element. Any number of elements may be assigned the same class name or names. Multiple class names must be separated by white space characters.
dir	Specifies the direction of directionally neutral text in an element's content and attribute values: **ltr** indicates left-to-right text or table. **rtl** indicates right-to-left text or table.
id	Assigns a name to an element. This name must be unique in a document.
lang	Specifies the base language of an element's attribute values and text content. The default value of this attribute is unknown.
noshade	If set to **true**, the horizontal rule is rendered in a solid color. If set to **false**, the rule is rendered as a two-color groove.
onclick	The onclick event occurs when the pointing device button is clicked over an element.
ondblclick	The ondblclick event occurs when the pointing device button is double clicked over an element.
onkeydown	The onkeydown event occurs when a key is pressed down over an element.
onkeypress	The onkeypress event occurs when a key is pressed and released over an element.
onkeyup	The onkeyup event occurs when a key is released over an element. This attribute may be used with most elements.
onmousedown	The onmousedown event occurs when the pointing device button is pressed over an element.
onmousemove	The onmousemove event occurs when the pointing device is moved while it is over an element.
onmouseout	The onmouseout event occurs when the pointing device is moved away from an element.
onmouseover	The onmousemove event occurs when the pointing device is moved while it is over an element.
onmouseup	The onmouseup event occurs when the pointing device button is released over an element.
size	Specifies the initial width of the control. The width is given in pixels except when **type** attribute is set to **text** or **password**. In that case, its value refers to the (integer) number of characters.

src	When the **type** attribute is set to **image**, this attribute specifies the location of the image to be used to decorate the graphical submit button.
style	Specifies style information for the current element. The syntax of the value of the style attribute is determined by the default style sheet language.
title	Specifies advisory information about the element for which it is set.
width	Specifies the width of the rule. The default width is 100%, which extends the rule across the entire canvas.

See Also

W3C HTML Home Page

3.2.7.5 IMG HTML Element

The IMG element embeds an image or a video clip in the document and provides the following attributes and events.

Item	Description
align	Specifies the horizontal alignment of its element with respect to the surrounding context. Possible values are **left**, **right**, **center**, and **justify**.
alt	For user agents that cannot display images, forms, or applets, this attribute specifies alternate text. The language of the alternate text is specified by the lang attribute.
border	Specifies the width of an image border, in pixels.
class	Assigns a class name or set of class names to an element. Any number of elements may be assigned the same class name or names. Multiple class names must be separated by white space characters.
dir	Specifies the direction of directionally neutral text in an element's content and attribute values: **ltr** indicates left-to-right text or table. **rtl** indicates right-to-left text or table.
height	Specifies the image and object height override.
hspace	Specifies the amount of white space to be inserted to the left and right of an image.
id	Assigns a name to an element. This name must be unique in a document.
ismap	For the IMG and INPUT elements, associates a server-side image map with the element.
lang	Specifies the base language of an element's attribute values and text content. The default value of this attribute is unknown.
longdesc	Specifies a link to a long description of the image.
name	Assigns the control name.
onclick	The onclick event occurs when the pointing device button is clicked over an element.
ondblclick	The ondblclick event occurs when the pointing device button is double clicked over an element.
onkeydown	The onkeydown event occurs when a key is pressed down over an element.
onkeypress	The onkeypress event occurs when a key is pressed and released over an element.
onkeyup	The onkeyup event occurs when a key is released over an element. This attribute may be used with most elements.
onmousedown	The onmousedown event occurs when the pointing device button is pressed over an element.
onmousemove	The onmousemove event occurs when the pointing device is moved while it is over an element.

onmouseout	The onmouseout event occurs when the pointing device is moved away from an element.
onmouseover	The onmousemove event occurs when the pointing device is moved while it is over an element.
onmouseup	The onmouseup event occurs when the pointing device button is released over an element.
src	Specifies the location of the image to be used in the control.
style	Specifies style information for the current element. The syntax of the value of the style attribute is determined by the default style sheet language.
title	Specifies advisory information about the element for which it is set.
usemap	Associates an image map with an element. The image map is defined by a MAP element. The value of usemap must match the value of the name attribute of the associated MAP element.
vspace	Specifies the amount of white space to be inserted above and below an image.
width	Specifies the image and object width override.

See Also

W3C HTML Home Page

3.2.7.6 INPUT HTML Element

The INPUT element specifies a form input control and provides the following attributes and events.

Item	Description
accept	Specifies a comma-separated list of content types that a server processing this form will handle correctly. User agents may use this information to filter non-conforming files when prompting a user to select files to be sent to the server.
accesskey	Assigns an access key to an element. An access key is a single character from the document character set. Authors should consider the input method of the expected reader when specifying an accesskey.
align	Specifies the horizontal alignment of its element with respect to the surrounding context. Possible values are **left**, **right**, **center**, and **justify**.
alt	For user agents that cannot display images, forms, or applets, this attribute specifies alternate text. The language of the alternate text is specified by the lang attribute.
checked	When the **type** attribute has the value **radio** or **checkbox**, this boolean attribute specifies that the button is on. User agents must ignore this attribute for other control types.
class	Assigns a class name or set of class names to an element. Any number of elements may be assigned the same class name or names. Multiple class names must be separated by white space characters.
dir	Specifies the direction of directionally neutral text in an element's content and attribute values: **ltr** indicates left-to-right text or table. **rtl** indicates right-to-left text or table.
disabled	When set for a form control, this boolean attribute disables the control for user input.
id	Assigns a name to an element. This name must be unique in a document.
ismap	For the IMG and INPUT elements, associates a server-side image map with the element.
lang	Specifies the base language of an element's attribute values and text content. The default value of this attribute is unknown.

maxlength	When the **type** attribute is set to **text** or **password**, this attribute specifies the maximum number of characters the user may enter. This number may exceed the specified size, in which case the user agent should offer a scrolling mechanism. The default value for this attribute is an unlimited number.
name	Assigns the control name.
onblur	The onblur event occurs when an element loses focus either by the pointing device or by tabbing navigation.
onchange	The onchange event occurs when a control loses the input focus and its value has been modified since gaining focus. This attribute applies to the following elements: INPUT, SELECT, and TEXTAREA.
onclick	The onclick event occurs when the pointing device button is clicked over an element.
ondblclick	The ondblclick event occurs when the pointing device button is double clicked over an element.
onfocus	The onfocus event occurs when an element receives focus either by the pointing device or by tabbing navigation.
onkeydown	The onkeydown event occurs when a key is pressed down over an element.
onkeypress	The onkeypress event occurs when a key is pressed and released over an element.
onkeyup	The onkeyup event occurs when a key is released over an element. This attribute may be used with most elements.
onmousedown	The onmousedown event occurs when the pointing device button is pressed over an element.
onmousemove	The onmousemove event occurs when the pointing device is moved while it is over an element.
onmouseout	The onmouseout event occurs when the pointing device is moved away from an element.
onmouseover	The onmousemove event occurs when the pointing device is moved while it is over an element.
onmouseup	The onmouseup event occurs when the pointing device button is released over an element.
onselect	The onselect event occurs when a user selects some text in a text field. This attribute may be used with the INPUT and TEXTAREA elements.
readonly	When set for a form control, this boolean attribute prohibits changes to the control.
size	Specifies the initial width of the control. The width is given in pixels except when **type** attribute is set to **text** or **password**. In that case, its value refers to the (integer) number of characters.
src	When the **type** attribute is set to **image**, this attribute specifies the location of the image to be used to decorate the graphical submit button.
style	Specifies style information for the current element. The syntax of the value of the style attribute is determined by the default style sheet language.
tabindex	Specifies the position of the current element in the tabbing order for the current document. This value must be a number between 0 and 32767. User agents should ignore leading zeros.
title	Specifies advisory information about the element for which it is set.

type	Specifies the type of control to create and can be set to: **button** creates a push button. User agents should use the value of the value attribute as the button's label. **checkbox creates a checkbox.** **file creates a file select control. User agents may use the value of the value attribute as the initial file name.** **hidden creates a hidden control.** **image creates a graphical submit button. The value of the src attribute specifies the URI of the image that will decorate the button. For accessibility reasons, authors should provide alternate text for the image via the alt attribute.** **password creates a single-line text input control, but the input text is rendered to hide the characters (for example, a series of asterisks). This control type is often used for sensitive input such as passwords. Note that the current value is the text entered by the user, not the text rendered by the user agent.** **radio creates a radio button.** **reset creates a reset button.** **submit creates a submit button.** **text creates a single-line text input control.**
usemap	Associates an image map with an element. The image map is defined by a MAP element. The value of usemap must match the value of the name attribute of the associated MAP element.
value	Specifies the initial value of the control. It is optional except when the **type** attribute has the value **radio** or **checkbox**.

See Also

W3C HTML Home Page

3.2.7.7 SELECT HTML Element

The SELECT element specifies a list box or dropdown list and provides the following attributes and events.

Item	Description
class	Assigns a class name or set of class names to an element. Any number of elements may be assigned the same class name or names. Multiple class names must be separated by white space characters.
dir	Specifies the direction of directionally neutral text in an element's content and attribute values: **ltr** indicates left-to-right text or table. **rtl** indicates right-to-left text or table.
disabled	When set for a form control, this boolean attribute disables the control for user input.
id	Assigns a name to an element. This name must be unique in a document.
lang	Specifies the base language of an element's attribute values and text content. The default value of this attribute is unknown.
multiple	If set to **true**, allows multiple selections in the list box or dropdown list. If set to **false**, the SELECT element only permits single selections.
name	Assigns the control name.
onblur	The onblur event occurs when an element loses focus either by the pointing device or by tabbing navigation.

onchange	The onchange event occurs when a control loses the input focus and its value has been modified since gaining focus. This attribute applies to the following elements: INPUT, SELECT, and TEXTAREA.
onclick	The onclick event occurs when the pointing device button is clicked over an element.
ondblclick	The ondblclick event occurs when the pointing device button is double clicked over an element.
onfocus	The onfocus event occurs when an element receives focus either by the pointing device or by tabbing navigation.
onkeydown	The onkeydown event occurs when a key is pressed down over an element.
onkeypress	The onkeypress event occurs when a key is pressed and released over an element.
onkeyup	The onkeyup event occurs when a key is released over an element. This attribute may be used with most elements.
onmousedown	The onmousedown event occurs when the pointing device button is pressed over an element.
onmousemove	The onmousemove event occurs when the pointing device is moved while it is over an element.
onmouseout	The onmouseout event occurs when the pointing device is moved away from an element.
onmouseover	The onmousemove event occurs when the pointing device is moved while it is over an element.
onmouseup	The onmouseup event occurs when the pointing device button is released over an element.
size	Specifies the initial width of the control. The width is given in pixels except when **type** attribute is set to **text** or **password**. In that case, its value refers to the (integer) number of characters.
style	Specifies style information for the current element. The syntax of the value of the style attribute is determined by the default style sheet language.
tabindex	Specifies the position of the current element in the tabbing order for the current document. This value must be a number between 0 and 32767. User agents should ignore leading zeros.
title	Specifies advisory information about the element for which it is set.

See Also

W3C HTML Home Page

3.2.7.8 SPAN HTML Element

The SPAN element lets you to define your own method of rendering, using style sheets and provides the following attributes and events.

Item	Description
class	Assigns a class name or set of class names to an element. Any number of elements may be assigned the same class name or names. Multiple class names must be separated by white space characters.
dir	Specifies the direction of directionally neutral text in an element's content and attribute values: **ltr** indicates left-to-right text or table. **rtl** indicates right-to-left text or table.
id	Assigns a name to an element. This name must be unique in a document.
lang	Specifies the base language of an element's attribute values and text content. The default value of this attribute is unknown.
onclick	The onclick event occurs when the pointing device button is clicked over an element.
ondblclick	The ondblclick event occurs when the pointing device button is double clicked over an element.

onkeydown	The onkeydown event occurs when a key is pressed down over an element.
onkeypress	The onkeypress event occurs when a key is pressed and released over an element.
onkeyup	The onkeyup event occurs when a key is released over an element. This attribute may be used with most elements.
onmousedown	The onmousedown event occurs when the pointing device button is pressed over an element.
onmousemove	The onmousemove event occurs when the pointing device is moved while it is over an element.
onmouseout	The onmouseout event occurs when the pointing device is moved away from an element.
onmouseover	The onmousemove event occurs when the pointing device is moved while it is over an element.
onmouseup	The onmouseup event occurs when the pointing device button is released over an element.
style	Specifies style information for the current element. The syntax of the value of the style attribute is determined by the default style sheet language.
title	Specifies advisory information about the element for which it is set.

See Also

W3C HTML Home Page

3.2.7.9 TABLE HTML Element

The TABLE element specifies that the contained content is organized into a table with rows and columns. Use the TR, TD, and TH elements in the container to create the rows, columns, and cells. The TABLE element provides the following attributes and events.

Tip: From the IDE main menu, choose View->Toolbars->HTML Table

to display a toolbar for formatting tables.

Item	Description
align	Specifies the horizontal alignment of its element with respect to the surrounding context. Possible values are **left**, **right**, **center**, and **justify**.
bgcolor	Sets the background color for the table cells.
border	Specifies the width (in pixels only) of the frame around a table.
cellpadding	Specifies the amount of space between the border of the cell and its contents. If the value of this attribute is a pixel length, all four margins should be this distance from the contents. If the value of the attribute is a percentage length, the top and bottom margins should be equally separated from the content based on a percentage of the available vertical space, and the left and right margins should be equally separated from the content based on a percentage of the available horizontal space.
cellspacing	Specifies how much space the user agent should leave between the left side of the table and the left-hand side of the leftmost column, the top of the table and the top side of the topmost row, and so on for the right and bottom of the table.
class	Assigns a class name or set of class names to an element. Any number of elements may be assigned the same class name or names. Multiple class names must be separated by white space characters.
datapagesize	Specifies the number of records displayed in a table bound to a data source.
dir	Specifies the direction of directionally neutral text in an element's content and attribute values: **ltr** indicates left-to-right text or table. **rtl** indicates right-to-left text or table.

frame	
id	Assigns a name to an element. This name must be unique in a document.
lang	Specifies the base language of an element's attribute values and text content. The default value of this attribute is unknown.
onclick	The onclick event occurs when the pointing device button is clicked over an element.
ondblclick	The ondblclick event occurs when the pointing device button is double clicked over an element.
onkeydown	The onkeydown event occurs when a key is pressed down over an element.
onkeypress	The onkeypress event occurs when a key is pressed and released over an element.
onkeyup	The onkeyup event occurs when a key is released over an element. This attribute may be used with most elements.
onmousedown	The onmousedown event occurs when the pointing device button is pressed over an element.
onmousemove	The onmousemove event occurs when the pointing device is moved while it is over an element.
onmouseout	The onmouseout event occurs when the pointing device is moved away from an element.
onmouseover	The onmousemove event occurs when the pointing device is moved while it is over an element.
onmouseup	The onmouseup event occurs when the pointing device button is released over an element.
rules	Specifies which rules will appear between cells within a table. The rendering of rules is user agent dependent. Possible values: **none** No rules. This is the default value. **groups** Rules will appear between row groups and column groups only. **rows** Rules will appear between rows only. **cols** Rules will appear between columns only. **all** Rules will appear between all rows and columns.
style	Specifies style information for the current element. The syntax of the value of the style attribute is determined by the default style sheet language.
title	Specifies advisory information about the element for which it is set.
width	Specifies the desired width of the entire table and is intended for visual user agents. When the value is a percentage value, the value is relative to the user agent's available horizontal space. In the absence of any width specification, table width is determined by the user agent.

See Also

W3C HTML Home Page

3.2.7.10 TEXTAREA HTML Element

The TEXTAREA element specifies a multi-line text input control and provides the following attributes and events.

Item	Description
accesskey	Assigns an access key to an element. An access key is a single character from the document character set. Authors should consider the input method of the expected reader when specifying an accesskey.
class	Assigns a class name or set of class names to an element. Any number of elements may be assigned the same class name or names. Multiple class names must be separated by white space characters.

3

cols	Specifies how many characters wide the text area is. Users should be able to enter longer lines than this, so user agents should provide some means to scroll through the contents of the control when the contents extend beyond the visible area. User agents may wrap visible text lines to keep long lines visible without the need for scrolling.
dir	Specifies the direction of directionally neutral text in an element's content and attribute values: **ltr** indicates left-to-right text or table. **rtl** indicates right-to-left text or table.
disabled	When set for a form control, this boolean attribute disables the control for user input.
id	Assigns a name to an element. This name must be unique in a document.
lang	Specifies the base language of an element's attribute values and text content. The default value of this attribute is unknown.
name	Assigns the control name.
onblur	The onblur event occurs when an element loses focus either by the pointing device or by tabbing navigation.
onchange	The onchange event occurs when a control loses the input focus and its value has been modified since gaining focus. This attribute applies to the following elements: INPUT, SELECT, and TEXTAREA.
onclick	The onclick event occurs when the pointing device button is clicked over an element.
ondblclick	The ondblclick event occurs when the pointing device button is double clicked over an element.
onfocus	The onfocus event occurs when an element receives focus either by the pointing device or by tabbing navigation.
onkeydown	The onkeydown event occurs when a key is pressed down over an element.
onkeypress	The onkeypress event occurs when a key is pressed and released over an element.
onkeyup	The onkeyup event occurs when a key is released over an element. This attribute may be used with most elements.
onmousedown	The onmousedown event occurs when the pointing device button is pressed over an element.
onmousemove	The onmousemove event occurs when the pointing device is moved while it is over an element.
onmouseout	The onmouseout event occurs when the pointing device is moved away from an element.
onmouseover	The onmousemove event occurs when the pointing device is moved while it is over an element.
onmouseup	The onmouseup event occurs when the pointing device button is released over an element.
onselect	The onselect event occurs when a user selects some text in a text field. This attribute may be used with the INPUT and TEXTAREA elements.
readonly	When set for a form control, this boolean attribute prohibits changes to the control.
rows	Specifies the number of visible text lines in the control. Users should be able to enter more lines than this, so user agents should provide some means to scroll through the contents of the control when the contents extend beyond the visible area.
style	Specifies style information for the current element. The syntax of the value of the style attribute is determined by the default style sheet language.
tabindex	Specifies the position of the current element in the tabbing order for the current document. This value must be a number between 0 and 32767. User agents should ignore leading zeros.
title	Specifies advisory information about the element for which it is set.

See Also

W3C HTML Home Page

3.2.8 Insert

Topics

Name	Description
Insert User Control (see page 813)	**Insert ▶Insert User Control** Use this dialog box to insert a user control into a Web Form or a User Control template.
Insert Image (see page 813)	**Insert ▶Image** Use this dialog box to insert an image file into your ASP.NET Web Form or HTML page. Once inserted, you can select and change the following image attributes.
Insert Input (see page 814)	**Insert ▶Input** Use this dialog box to create and modify controls on your ASP.NET Web Form or HTML page before submitting it for deployment.
Insert Table (see page 814)	**Insert ▶Table** Use this dialog box to insert a table into your ASP.NET Web Form or HTML page. Once inserted, you can modify the physical appearance of the table.
Color Selector (see page 815)	Use this dialog box to change the foreground and background color from the HTML Designer.

3.2.8.1 Insert User Control

Insert ▶Insert User Control

Use this dialog box to insert a user control into a Web Form or a User Control template.

Item	Description
Browse	Search for the .ascx file in a directory.
OK	Add the .ascx file to your project.
Cancel	Exits the dialog without saving.

3.2.8.2 Insert Image

Insert ▶Image

Use this dialog box to insert an image file into your ASP.NET Web Form or HTML page. Once inserted, you can select and change the following image attributes.

Item	Description
Alignment	Specifies the position of the image or object with respect to its context.
Border size	Specifies the width of an image or object border in pixels.
Width	Specifies a new width for the image or object selected.
Horizontal spacing	Specifies the amount of white space to be inserted to the left and right of an image or object.
Height	Specifies a new height for the image or object selected.
Vertical spacing	Specifies the amount of white space to be inserted above and below an image or object.
Alternate text	Specify alternate text to serve as content when the element cannot be rendered normally.

Image map	Specifies a client-side image map (or other navigation mechanism) that may be associated with another element.
Image map information on server	Defines a server-side image map for the specified image.

3.2.8.3 Insert Input

Insert ▶Input

Use this dialog box to create and modify controls on your ASP.NET Web Form or HTML page before submitting it for deployment.

Item	Description
Input type	Specifies the type of controls to create.
Name	Specifies a name of the form.
Alignment	Specifies the horizontal alignment of its element with respect to the surrounding context.
Value	Specifies the initial value of the control.
Alt text	Specifies alternative text when the element cannot be rendered normally, such as images or applets.
Tab Index	Specifies the position of the current element in the tabbing order for the current document. Values should be between 0 and 32767.
Read only	Prohibits changes to the control.
Max length	Specifies the maximum number of characters you may enter if the attribute has the text or password value.

3.2.8.4 Insert Table

Insert ▶Table

Use this dialog box to insert a table into your ASP.NET Web Form or HTML page. Once inserted, you can modify the physical appearance of the table.

Item	Description
Rows	Specifies the number of visible text lines.
Columns	Specifies the number of columns that appear in the table.
Width	Specifies the desired width of the entire table.
pixels	Specifies an integer value that represents the number of pixels on the screen.
percent	Specifies a value in %.
Height	Specifies the desired height of the entire table in pixels.
Alignment	Specifies the position of the table with respect to the document.
Background color	Sets the background color for the table.
Background Image	Sets the background image for the table.
Border color	Sets the color of the surrounding frame of the table.
Border size	Specifies the width (in pixels only) of the table frame.

Highlight color	Specifies the color used to draw the light portion of the table frame.
Cell spacing	Specifies how much space you should leave between the left side of the table and the left-hand side of the leftmost column, the top of the table and the top side of the topmost row, and so on for the right and bottom of the table. The attribute also specifies the amount of space to leave between cells.
Shadow color	Specifies the color used to draw the shaded portion of the table frame.
Cell padding	Specifies the amount of space between the border of the cell and its contents.
Summary	Provides a summary of the table's purpose and structure to render non-visual media.
Background Color for Cell Attributes	Sets the background color for the individual cells within a table.
Border color for Cell Attributes	Sets the color of the surrounding frame of individual cells within a table.
Text Alignment	Specifies the alignment of data and the justification of text in a cell.
Wrap text	Check this box to specify the text to be wrapped in the same width as the table.

3.2.8.5 Color Selector

Use this dialog box to change the foreground and background color from the HTML Designer.

Item	Description
Web palette	Use this tab to select from a variety of web colors available.
Use color name	Select to use either a web color string, i.e., #FF00FF, or a web color name like "snow." If checked, the dialog will use the web color name when available.
Customize Selected Color	Lets you specify and modify the values of the color scheme to increase or decrease the composition of the selected color.
Named Web Colors	Use this tab to easily select a named web color.
Other Colors	Use this tab to select from a list of system or standard colors.
System Colors	Lists available colors from the Windows system palette.
Standard Colors	Lists of available colors from the Windows standard 16 color palette.

3.2.9 Testing Wizards

Topics

Name	Description
Unit Test Case Wizard (see page 816)	**File ▶ New ▶ Other ▶ Unit Test ▶ Test Case** Use this wizard to create a test case for your current project.
Unit Test Case Wizard (see page 816)	**File ▶ New ▶ Other ▶ Unit Test ▶ Test Case** Use this wizard page to supply the details for the test case you want to create.
Unit Test Project Wizard (see page 816)	**File ▶ New ▶ Other ▶ Unit Tests ▶ Test Project** Use this wizard to create a test suite that includes a number of tests.
Unit Test Project Wizard (see page 817)	**File ▶ New ▶ Other ▶ Unit Tests ▶ Test Project** Use this wizard to specify the framework and test runner for your test project.

3.2.9.1 Unit Test Case Wizard

File ▶ New ▶ Other ▶ Unit Test ▶ Test Case

Use this wizard to create a test case for your current project.

Item	Description
Source File	Specifies the path and file name of the code file to which you want to add tests. By default, this textbox is prefilled with the name of the primary code file of the active project. You can change the location and the name of the file to a file other than the default. If the file does not exist, you will be unable to select classes and methods to test and will be unable to complete the wizard. For example, if you add a unit to your project and you want to add test cases to the unit, you must add at least one class to the unit to be able to use this wizard successfully.
Available Classes and Methods	Displays a treeview of the available classes and methods for the current file. You can deselect items in the tree. By default, all classes and methods are selected. If you deselect individual methods within a class, the wizard ignores those methods when building test cases. If you deselect the class, the wizard ignores the entire class and all of its methods, even if you do not deselect the methods. If you select a class but do not select any methods in that class, the wizard generates a test case for the class, but does not generate any test methods for that class.

See Also

Unit Testing Overview (see page 70)

3.2.9.2 Unit Test Case Wizard

File ▶ New ▶ Other ▶ Unit Test ▶ Test Case

Use this wizard page to supply the details for the test case you want to create.

Item	Description
Test Project	Specifies the name of the project. Prefilled with the default, which is the current project.
File Name	Specifies the name of the code file containing the classes and methods you want to test. Prefilled with the default, which is the current code file.
Test Framework	Specifies the testing framework you want to use. RAD Studio auto-detects the type of project you are testing and sets this to the correct framework that is currently supported for that code personality.
Base Class	Specifies the base class to be inherited by the test. By default, the test case is built using the base class of the active code file. Optional.

See Also

Unit Testing Overview (see page 70)

3.2.9.3 Unit Test Project Wizard

File ▶ New ▶ Other ▶ Unit Tests ▶ Test Project

Use this wizard to create a test suite that includes a number of tests.

Item	Description
Project Name	Specify the name of the project or accept the default, which is the name of the active project.

Location	Specify the path for the project file.
Personality	Select a personality. Prefilled based on the type of the active project.
Add to project group	Select this if you want to add the new Test Project to your current project group. By default, this is selected. If you are creating a test project to be used in multiple projects, you might opt to deselect this option.

See Also

Unit Testing Overview (see page 70)

3.2.9.4 Unit Test Project Wizard

File ▶ New ▶ Other ▶ Unit Tests ▶ Test Project

Use this wizard to specify the framework and test runner for your test project.

Item	Description
Test Framework	Specify the framework you want to use to build the test project. By default, this is set to **NUnit** for the Delphi for .NET and C# personalities; for Delphi for Win32 and C++ this can only be DUnit. For the C# personality, only NUnit is supported.
Test Runner	Specify the test runner you want to use for running the test project. The GUI Test Runner is selected by default. You can also choose the Console Test Runner, which causes the test project assembly to contain a command to execute the console test runner from the framework directory.

See Also

Unit Testing Overview (see page 70)

3.2.10 NET_VS

Topics

Name	Description
Advanced Data Binding (see page 818)	This dialog is a component of the Microsoft .NET Framework. Use this dialog to bind a property to a value from a valid data provider to a Windows Form. For more information, refer to the documentation resources on the Microsoft Developer Network (MSDN).
AutoFormat (see page 818)	This dialog is a component of the Microsoft .NET Framework. Use this dialog to apply a predefined format, including borders, colors, and fill patterns, to a tabular control. For more information, refer to the documentation resources on the Microsoft Developer Network (MSDN).
Collection Editor (see page 818)	This dialog is a component of the Microsoft .NET Framework. Use this dialog to create and edit individual members of a collection. For more information, refer to the documentation resources on the Microsoft Developer Network (MSDN).
Databindings (see page 818)	This dialog is a component of the Microsoft .NET Framework. Use this dialog to bind to a data item and set formatting. For more information, refer to the documentation resources on the Microsoft Developer Network (MSDN).
Dynamic Properties (see page 818)	Use this dialog to set dynamic properties that can be changed in a configuration file, without recompiling the application.

Properties (see page 819)	This dialog is a component of the Microsoft .NET Framework. Use this dialog to set properties for the currently selected object. For more information, refer to the documentation resources on the Microsoft Developer Network (MSDN).

3.2.10.1 Advanced Data Binding

This dialog is a component of the Microsoft .NET Framework.

Use this dialog to bind a property to a value from a valid data provider to a Windows Form.

For more information, refer to the documentation resources on the Microsoft Developer Network (MSDN).

3.2.10.2 AutoFormat

This dialog is a component of the Microsoft .NET Framework.

Use this dialog to apply a predefined format, including borders, colors, and fill patterns, to a tabular control.

For more information, refer to the documentation resources on the Microsoft Developer Network (MSDN).

3.2.10.3 Collection Editor

This dialog is a component of the Microsoft .NET Framework.

Use this dialog to create and edit individual members of a collection.

For more information, refer to the documentation resources on the Microsoft Developer Network (MSDN).

3.2.10.4 Databindings

This dialog is a component of the Microsoft .NET Framework.

Use this dialog to bind to a data item and set formatting.

For more information, refer to the documentation resources on the Microsoft Developer Network (MSDN).

3.2.10.5 Dynamic Properties

Use this dialog to set dynamic properties that can be changed in a configuration file, without recompiling the application.

Item	Description
Properties	Check the properties that you want to add to the configuration file.
Key mapping	Optionally, override the default key for the property that appears in the configuration file. This is useful if you want to provide a more descriptive key for the property.

Note: Property values stored in a configuration file are not secure. Confidential information, such as passwords, should not be stored as dynamic properties.

See Also

Introduction to Dynamic Properties

Using Dynamic Properties (↗ see page 161)

3.2.10.6 Properties

This dialog is a component of the Microsoft .NET Framework.

Use this dialog to set properties for the currently selected object.

For more information, refer to the documentation resources on the Microsoft Developer Network (MSDN).

3.2.11 Project

Topics

Name	Description
ASP.NET Deployment Manager (↗ see page 821)	To open the **Deployment Manager**, right-click the **Deployment** node in the **Project Manager**, and then choose the **New ASP.NET Deployment** option. Use this dialog box to automatically collect all of the .aspx, .asax, web.config, and related assembly files that are needed to deploy your ASP.NET or IntraWeb project.
Project Options (↗ see page 822)	
COM Imports (↗ see page 846)	**Project ▶ Add Reference** Adds a COM type library to the current project. The **Add Reference** dialog box is also available in the **Project Manager** by right-clicking a **References** folder and choosing Add Reference.
C++ Project Options (↗ see page 847)	
.NET Assemblies (↗ see page 900)	**Project ▶ Add Reference** Adds a .NET assembly reference to the current project. The **Add Reference** dialog box is also available in the **Project Manager** by right-clicking a **References** folder and choosing Add Reference.
Project References (↗ see page 901)	**Project ▶ Add Reference** Adds a reference to a project that produces an assembly (.dll), such as a Class Library or Control Library. The reference will be added to the current project. The **Add Reference** dialog box is also available in the **Project Manager** by right-clicking a **References** folder and choosing Add Reference.
Add to Repository (↗ see page 901)	**Project ▶ Add to Repository** Saves a customized form or project template in the **Object Repository** for reuse in other projects. The saved forms and templates are then available in the **New Items** dialog box when you choose **File ▶ New ▶ Other**.
UDDI Browser (↗ see page 902)	**Project ▶ Add Web Reference** Searches for services and providers in the UDDI services sites with WSDL described services. Search by name or browse through available categorization schemas.
Change Package (↗ see page 902)	Adds required units to your package. This dialog box appears when the **Package Editor** tries to compile a package and detects that the package cannot be built, or is incompatible with another package currently loaded by the IDE. This occurs because the package uses one or more units that are found in another package.
Project Dependencies (↗ see page 903)	**Project ▶ Dependencies** Creates project dependencies within a project group. From the list, choose the projects to build before building the selected project.
Add Languages (↗ see page 903)	**Project ▶ Languages ▶ Add Language** Adds one or more language resource DLLs to a project. Follow the instructions on each wizard page.
Remove Language (↗ see page 903)	**Project ▶ Languages ▶ Remove Language** Removes one or more languages from the project. Follow the instructions on each wizard page.

Set Active Language (↗ see page 903)	**Project ▶ Languages ▶ Set Active Language** Determines which language module loads when you run your application in the IDE. Before changing the active language, make sure you have recompiled the satellite assembly for the language you want to use. Select the desired language and click **Finish**.
New Category Name (↗ see page 904)	**Tools ▶ Repository ▶ Edit button ▶ New Category button** Use this dialog box to assign a name to a new category in the **Object Repository**.
Information (↗ see page 904)	**Project ▶ Information** Views the program compilation information and compilation status for your project.
Project Page Options (↗ see page 904)	**Project ▶ Project Page Options** Specifies an HTML file in your project as the Project Page for recording a description of the project, and various other notes and information. This page is automatically displayed in the IDE when you open the project.
Remove from Project (↗ see page 904)	**Project ▶ Remove from Project** Removes one or more files from the current project.
Options (↗ see page 905)	**Project Manager ▶ Right-click a satellite assembly ▶ Options command** Sets the assembly linker (`al.exe`) options for the satellite assembly selected in the **Project Manager**.
Select Icon (↗ see page 906)	Selects a bitmap to represent your template in the **New Items** dialog box. You can use a bitmap of any size, but it will be cropped to 60 x 40 pixels.
Web Deploy Options (↗ see page 906)	**Project ▶ Web Deploy Options** Configures a finished ActiveX control or ActiveForm for deployment to a Windows Web server. **Tip:** Set these options before you compile the ActiveX project and deploy it by choosing Project->Web Deploy.
Build All Projects (↗ see page 907)	**Project ▶ Build All Projects** Compiles all of the source code in the current project group, regardless of whether any source code has changed. Building is useful when you are unsure which files have changed, or if you have changed project or compiler options.
Build Project (↗ see page 908)	**Project ▶ Build Project** Rebuilds all files in your current project regardless of whether they have changed. Building is useful when you are unsure which files have changed, or if you have changed project or compiler options.
Compile and Make All Projects (↗ see page 908)	**Project ▶ Compile** **Project ▶ Make All Projects** Compile (for Delphi) and Make (for C++) compiles only those files that have changed since the last build, as well as any files that depend on them. Compiling or making does not execute the application (see **Run ▶ Run**).
Add to Project (↗ see page 908)	**Project ▶ Add to Project** Adds another source file to an already open project.
Add New Project (↗ see page 909)	**Project ▶ Add New Project** Adds new projects via the **New Items** dialog box .
Clean Package (↗ see page 909)	**Project ▶ Clean Package** Removes previously compiled files and leaves behind only the source files needed to build the project. Specifically , it cleans out any .dcu's, .bpl's, etc., that were generated.
Default Options (↗ see page 909)	**Project ▶ Default Options** Opens the default **Project Options** dialog box for the specified project type: C++ Builder, Delphi for Win32, Delphi for .NET, C# Builder, and Basic Builder. This option is only available when there is not an open project. After the **Project Options** dialog opens help is available from each page of the dialog. Click **Help** or press `F1`.
Options (↗ see page 909)	**Project ▶ Options** Opens the **Project Options** dialog that manages application and compiler options for your project. Making changes to your project only affects the current project. However, you can also save your selections as the default settings for new projects.
Syntax Check for Project (↗ see page 910)	**Project ▶ Syntax Check for Project** Checks the active project for incorrect symbols. This is an efficient way to check and see if a large project will build correctly prior to compiling. Errors are reported in the **Compiling** dialog with details shown in the **Messages** pane.

Update Localized Projects (see page 910)	**Project ▶Languages ▶Update Localized Projects** Updates resource modules. When you add an additional resource, such as a button on a form, you must update your resource modules to reflect your changes. Build and save your project before you update the resources.
View Source (see page 910)	**Project ▶View Source** Shows the source of the project file that manages the running of the application. Dephi and Delphi .NET show a .dpr file. C++ shows .cpp file.

3.2.11.1 ASP.NET Deployment Manager

To open the **Deployment Manager**, right-click the **Deployment** node in the **Project Manager**, and then choose the **New ASP.NET Deployment** option. Use this dialog box to automatically collect all of the .aspx, .asax, web.config, and related assembly files that are needed to deploy your ASP.NET or IntraWeb project.

Deployment Page

Specifies file locations and status of Source and Destination files for your ASP.NET deployment.

Item	Description
Source Directory	Displays the directory where your project is located.
Destination	Choose a target for your deployment, either a folder location or an FTP location. The target is the directory where your executable, .dlls, mark-up files, configuration, and other files that are necessary to run your application will be located. • If you choose Folder Location, the **Browse for Folder** dialog box is displayed. To change a folder location you have chosen, click the button with the ellipsis label (...) to browse again. • If you choose FTP Location, the **FTP Connection Options** dialog box is displayed.
Source Files	Lists most of the files that you need to include in your deployment directory (such as mark-up files, executables, and config files). **Note:** You may need to open the References node for your project (in the Project Manager) and set the Copy Local option to **True** for some of the .dll files, before you can add them to your project.. (For instance, if you're using BDP, you will have to do this with the Borland.Data.Common.dll and Borland.Data.Provider.dll files.) After you have set the **Copy Local** option, recompile your project. Now these additional files will show up in the **Source** list. To add database-specific drivers, right-click the References node for your project, and then select the Add References option. Choose the driver that corresponds to the database you are using.
Status	Displays the status of the files that you are going to deploy. Before you deploy the files, their satus is **Not Connected**. After you select a target destination, the status of these files will change to **New**.
Destination Files	Specifies the target destination for your application deployment and displays files that are already present at that destination. To move files from the Source Files list to the Destination Files list, right-click the file name, and choose either Copy Selected File(s) to Destination, or Copy All New and Modified Files to the Source Destination. You can also use the icons in the center gutter to copy files to and delete files from the destination list.
Status	Displays the status of the files that you are deploying. After you deploy your application, the files are listed with a status of **Current**.

Deployment Listbox Context Menu

When you right-click the Deployment Manager, you see the following deployment options.

Item	Description
Refresh	Re-display the lists, after changes have been made.

Copy Selected File(s) to Destination	Copy all of the specified Source files to the Destination directory.
Delete Selected Destination File(s)	Delete the selected files from the Destination list.
Change Destination Filename	Change the name of a Destination file.
Copy All New and Modified Files to Destination	Copy all of the specified Source files to the Destination directory.
Delete All Destination Files Not in Project	Delete all files from the destination list, that are not included in the project.
Show Assembly References	When selected, the Deployment Manager shows all of the assemblies referenced by the project. The system assemblies are shown, but disabled (grayed). These disabled assemblies can't be deployed.
External Files...	Allows you to pick the external files that you want to deploy. The External Files dialog box appears, with a list of check boxes. This list includes the BDP database libraries and the database-specific libraries, since these often need to be deployed with your ASP.NET application. You can add more files to the list using a **File Open** dialog. The list has a column that indicates the destination subdirectory for each external file. You can edit this destination. Check the boxes next to the libraries that you want to add to your application deployment, and then click the Add button.
Show Ignored Groups and Files	Display group and files that you have previously chosen to ignore.
Ignore Groups	Choose groups to filter out of the reference list.
Ignore Files	Choose files to filter out of the reference list.
Enable Logging	Create a log of your application deployment.
View Log	View the log created during the deployment of your project.

See Also

Deploying ASP.NET Applications

Using the ASP.NET Deployment Manager

FTP Connection Options Dialog Box (↗ see page 778)

3.2.11.2 Project Options

Topics

Name	Description
Configure Virtual Directory (↗ see page 824)	**Project ▶ Options ▶ ASP NET** Use this dialog box to specify and configure your server and its virtual root directory for your ASP.NET application.
Configure Cassini (↗ see page 824)	**Project ▶ Options ▶ Debugger ▶ ASP NET** Use this dialog box to specify a path and port number to the Cassini Web Server. The dialog appears only when you have not configured Cassini using **Tools ▶ Options ▶ ASP NET Options**.
ASP .NET (↗ see page 825)	**Project ▶ Options ▶ Debugger ▶ ASP NET** Use this dialog box to setup debugger options for your ASP.NET applications.
Add Design Packages (↗ see page 825)	**Project ▶ Options ▶ Packages ▶ Design packages Add button** Use this dialog box to navigate to a design time package and add it to the **Design packages** list.
Add Runtime Packages (↗ see page 825)	**Project ▶ Options ▶ Packages ▶ Ellipsis button ▶ Ellipsis button** Use this dialog box to add a runtime package to the **Design packages** list.

Add Symbol Table Search Path (see page 826)	Project ▶ Options ▶ Symbol Table ▶ New Use this dialog box to specify a new module name and path to be added to the list of symbols tables to be used during debugging. The module name and path are added to the list on the Symbol Tables page of the **Project Options** dialog box.
Application (see page 826)	Project ▶ Options ▶ Application Use this dialog box to change the name and type of the current Delphi for .NET application.
Application (see page 827)	Project ▶ Options ▶ Application Use this dialog box to change the name and type of the current application. **Note:** Not all of the options described below are available for all types of projects. For instance, the LIB attribute options are not available for all projects.
Apply Option Set (see page 828)	Project ▶ Options ▶ Load... Use this dialog to apply a named option set to a project configuration.
Build Configuration Manager (see page 828)	Project ▶ Configuration Manager Applies a named build configuration to a specific project or projects.
Build Events (see page 829)	Project ▶ Options ▶ Build Events Use the **Build Events** dialog box to add events for the pre-build, pre-link and post-build stages. Results of the commands you specify in this dialog box are displayed in the Output pane. To control the level of output, choose Tools ▶ Options ▶ Environment Options and adjust the **Verbosity** level. **Note:** Some of the build options are not available for some project types.
Compiler (see page 829)	Project ▶ Options ▶ Compiler Use this dialog box to set the C# compiler options for the current project.
Compiler (see page 831)	Project ▶ Options ▶ Compiler Use this page to set the compiler options for the current project. **Note:** Not all of the options described below are available for all types of projects.
Compiler Messages (see page 835)	Project ▶ Options ▶ Compiler Messages Use this page to control the information that the compiler provides at compile time.
Compiler (Visual Basic) (see page 836)	Project ▶ Options ▶ Compiler Use this dialog box to set the Visual Basic compiler options for the current project. For additional information about the Visual Basic compiler options, see the .NET Framework SDK online Help.
Components (see page 836)	Project ▶ Options ▶ Packages ▶ Components button Use this dialog box to display the components in the selected package, along with the icons that represent the components in the **Tool Palette** if the package is installed.
Debugger (see page 836)	Project ▶ Options ▶ Debugger Use this dialog to pass command-line parameters to your application, specify a host executable for testing a DLL, or load an executable into the debugger. This dialog is also available from Run ▶ Parameters.
Description (see page 837)	Project ▶ Options ▶ Description Use the **Description** page to specify a description for the package, the uses of the package, and how the package is built. **Note:** This page appears only if you are developing a package.
Directories/Conditionals (see page 838)	Project ▶ Options ▶ Directories/Conditionals Use this page to set your directory and conditional defines paths for C# and VB.
Directories/Conditionals (see page 838)	Project ▶ Options ▶ Directories/Conditionals Use this page to set your directory and conditional defines paths. You can also set the namespace prefix, to simplify your `uses` clause.
Debugger Environment Block (see page 839)	Project ▶ Options ▶ Debugger ▶ Environment Block Use this page to indicate which environment variables are passed to your application while you are debugging it. This page is also available by choosing Run ▶ Parameters.
Forms (see page 840)	Project ▶ Options ▶ Forms Use this page to select the main form for the current VCL Forms project and to choose which of the available forms are automatically created when your application begins.

Linker (↗ see page 840)	**Project ▶ Options ▶ Linker** Use this page to set linker options for your application. **Note:** Not all of the options described below are available for all types of projects.
Project Options (↗ see page 842)	**Project ▶ Options** Use this dialog box to set project options. In the left pane, click an item in a category to display its option page. If you leave the cursor over text describing an option, a tool tip gives you an option description, its default value, and a switch for the option if one exists. Each configuration, except the **Base**, is based on another configuration. In the **Project Manager** pane, the **Build Configurations** node under the project represents the **Base** configuration. All configurations are listed under the **Build Configurations** node in a hierarchical list that shows the parent-child relationship... more (↗ see page 842)
Packages (↗ see page 843)	**Project ▶ Options ▶ Packages** Use this page to specify the design time packages installed in the IDE and the runtime packages required by your project.
Pre-Build, Pre-Link, or Post-Build Events (↗ see page 844)	Use this dialog box to create a list of commands and macros to execute at certain points in the build process. Enter any valid list of DOS commands. The commands and their results are displayed in the on the **Output** tab of the **Messages** pane. **Project ▶ Options ▶ Build Events ▶ Edit...** **Note:** Pre-Link events are available only for C++ projects.
Signing (↗ see page 844)	**Project ▶ Options ▶ Signing** Use the Signing dialog to sign the assembly produced by the project using the Microsoft Strong Name (SN) utility. **Note:** Signing can only be used in the .NET framework.
Debugger Symbol Tables (↗ see page 845)	**Project ▶ Options ▶ Debugger ▶ Symbol Tables** Use this dialog box to specify the location of the symbols tables to be used during debugging. This dialog is also available from **Run ▶ Parameters**.
Version Info (↗ see page 845)	**Project ▶ Options ▶ Version Info** Use this dialog box to specify version information for a Delphi Win32 project. When version information is included, a user can right-click the program icon and select properties to display the version information.

3.2.11.2.1 Configure Virtual Directory

Project ▶ Options ▶ ASP NET

Use this dialog box to specify and configure your server and its virtual root directory for your ASP.NET application.

Item	Description
Location	Specify the root directory of your ASP.NET application.
Alias	Specify the name used to gain access to the virtual directory.
Read	Allows the Web Server to execute scripts.
Run scripts	Allows application execution for ISAPI or CGI type applications.
Execute	Allows you to upload files to the directory.
Write	Check this box to enable Write permissions.
Browse	Allows you to access list of files and directories.

3.2.11.2.2 Configure Cassini

Project ▶ Options ▶ Debugger ▶ ASP NET

Use this dialog box to specify a path and port number to the Cassini Web Server. The dialog appears only when you have not configured Cassini using **Tools ▸ Options ▸ ASP NET Options**.

Item	Description
Path	Indicate the path to the Cassini Web Server executable. To search for the executable file, click the elipse (...).
Port	Indicate the TCP/IP port used by the Cassini Web server.

Note: If you do not have access to the Cassini Web Server, you can download the server from http://www.asp.net/Projects/Cassini/Download.

3.2.11.2.3 ASP .NET

Project ▸ Options ▸ Debugger ▸ ASP NET

Use this dialog box to setup debugger options for your ASP.NET applications.

Item	Description
Launch Browser	Select the check box to automatically launch a web browser when the **Run** or **Run without Debugging** command is executed.
Start Page	Display the .asp file to be launched from a web browser.
HTTP Address	Specify the web location of the specified .asp file.
Host with Web Server	Select the check box to run the application under a web server when the **Run** or **Run without Debugging** command is executed.
Server	Indicate the default Web Server for new Web applications.
Virtual Directory	Specify the application root directory.
Server Options	Display the **Configure Virtual Directory** dialog box.
Default	Saves the current project settings as the default settings for new projects.

3.2.11.2.4 Add Design Packages

Project ▸ Options ▸ Packages ▸ Design packages Add button

Use this dialog box to navigate to a design time package and add it to the **Design packages** list.

3.2.11.2.5 Add Runtime Packages

Project ▸ Options ▸ Packages ▸ Ellipsis button ▸ Ellipsis button

Use this dialog box to add a runtime package to the **Design packages** list.

Item	Description
Package Name	Type the name of the package to add to the **Runtime Packages** list, or click the **Browse** button to search for the package using the **Package File Name** dialog. If the package is in the **Search Path**, a full path name is not required. (If the package directory is not in the **Search Path**, it is added at the end.)

Search Path	If you haven't included a full directory path in the **Package Name** edit box, make sure the directory where your package resides is in this list. If you add a directory in the **Search Path** edit box, you are changing the global Library Search Path. Click [...] to get an ordered list of search paths that you can edit. You can also change the path list in this field.

3.2.11.2.6 Add Symbol Table Search Path

Project ▶ Options ▶ Symbol Table ▶ New

Use this dialog box to specify a new module name and path to be added to the list of symbols tables to be used during debugging. The module name and path are added to the list on the Symbol Tables page of the **Project Options** dialog box.

Item	**Description**
Module Name	The name of the module to be added to the list of symbol tables. If the **Load all symbols** option is checked, all symbols are loaded and the list of modules is ignored. If the **Load all symbols** option is not checked, you can add the names of modules that the debugger will look for when loading symbol tables. Note that if you specify Load unspecified options, then only those in that list are not loaded. The string is interpreted as a module file name (such as **foo.dll**). This string can contain wildcards to specify multiple modules. For example, you can specify ***core*.bpl** to indicate modules such as **oldcore1.bpl**, **newcore2.bpl**, and so on.
Symbol Table Path	One or more directories containing the module to be added to the list of symbol tables. If you specify multiple directories for a module, use a semicolon to separate them. Click the **...** button to display the **Directory List** dialog, where you can choose a directory.

See Also

Overview of Debugging (↗ see page 10)

Setting the Search Order for Debug Symbol Tables (↗ see page 129)

Symbol Tables (↗ see page 845)

3.2.11.2.7 Application

Project ▶ Options ▶ Application

Use this dialog box to change the name and type of the current Delphi for .NET application.

Item	**Description**
Debug/Release	Indicates the current set of project options. By default, the distributed Debug and Release option sets have settings appropriate for debugging and deploying an application, respectively. The option sets provide an easy way to change project options based on your development activity. To create a user-defined option set, use the **Save as** button.
Save as	Displays a dialog box for naming and saving a user-defined set of project options.
Delete	Deletes the current option set. Only user-defined option sets can be deleted. The distributed Debug and Release option sets can not be deleted.
Windows executable	Creates a Windows executable next time you run the application. This option does not alter your code.
Console executable	Creates a console executable next time you run the application. This option does not alter your code.
Assembly	Creates an assembly next time you run the application. This option does not alter your code.

Application Name	Indicates the current name of the executable. By default, the project name is used as the executable name. To change the name, enter a new name in this field.
Startup object	Specify the class that contains the Main method to be used as the entry point for the program. This is useful is if your program contains more than one class with a Main method.
Default namespace	Specify the default namespace to be used for items that you add to the project by using the **New Items** dialog box. By default, the project name (without the extension) is used as the default namespace.
Application icon	Specify an icon (.ico) file to be inserted into the output file. The icon will be displayed next to the output file in Windows Explorer.

3.2.11.2.8 Application

Project ▶ Options ▶ Application

Use this dialog box to change the name and type of the current application.

Note: Not all of the options described below are available for all types of projects. For instance, the LIB attribute options are not available for all projects.

Application settings	Description
Title	Specify a title to appear next to the application icon when the application is minimized. The character limit is 255 characters.
Help file	Specify the location of the help file for the given application. Click the **Browse...** button to display an **Application Help File** dialog.
Icon	Specify an icon (.ico) file to be inserted into the output file. The icon is displayed next to the output file in Windows Explorer. Click the **Load Icon...** button to display an **Application Icon** dialog. This option corresponds to the /win32icon C# compiler option.
Enable runtime themes	Specifies that the application you are developing is to use runtime themes as for Windows Vista. The default value is true for preexisting projects and false for new projects.

Output settings	Description
Target file extension	Specifies the extension that is applied to the final executable file.

Library name settings	Description
LIB prefix	Adds the specified prefix to the DLL or package output file name.
LIB suffix	Adds the specified suffix to the DLL or package output file name before the extension.
LIB version	Adds a second extension to the DLL or package output file name after the extension. For example, if you specify 1.1.3 as the version for a DLL named WebApp, the output file is named WebApp.dll.1.1.3.

General option	Description
Default	Saves the current settings as the default for each new project.

3.2.11.2.9 Apply Option Set

Project ▶ Options ▶ Load...

Use this dialog to apply a named option set to a project configuration.

Item	Description
Option Set File	Navigates to the named option set. file. Click [...] to display a file selection dialog.
Action	You have three choices for how you apply values: **Overwrite**, **Replace**, or **Preserve**. **Overwrite** replaces the current configuration with the values from the option set entirely. That is, the values from the option set overwrite the current configuration and all other option values are set to their defaults. You get exactly the same configuration as when the option set was originally saved. **Replace** writes all the values from the option set to the current configuration, but no other values are changed. The values in the option set replace the current values, giving priority to the option set. **Preserve** writes only the values from the option set that are not already set in the active configuration. If the active configuration changed any of the values from their default values, they are not changed. This gives priority to the configuration.

See Also

Working with Named Option Sets (↗ see page 113)

3.2.11.2.10 Build Configuration Manager

Project ▶ Configuration Manager

Applies a named build configuration to a specific project or projects.

Item	Description
Configuration Name	Displays the names of the named build configurations. The two default configurations (**Debug** and **Release**) are listed, along with the build configurations you have created and named on the Project ▶ Options dialog box.
Available Projects	Lists the names of your projects, the active configuration associated with each project, and the path to the project. To apply the named build configuration listed in **Configuration Name**, select a project or projects and click **Apply**.

Tip: To create a named build configuration, use the Project->Options

dialog box. You can save options on several pages of this dialog box to a named configuration. The **Compiler**, **Compiler Messages**, **Linker**, and **Directories/Conditionals** pages each contain a **Configuration** field.

See Also

MSBuild Overview (↗ see page 4)

Build Configurations Overview (↗ see page 5)

Creating Named Build Configurations for C++ (↗ see page 107)

Creating Named Build Configurations in Delphi (↗ see page 108)

Applying the Active Build Configuration (↗ see page 104)

3.2.11.2.11 Build Events

Project ▶ Options ▶ Build Events

Use the **Build Events** dialog box to add events for the pre-build, pre-link and post-build stages. Results of the commands you specify in this dialog box are displayed in the Output pane. To control the level of output, choose **Tools ▶ Options ▶ Environment Options** and adjust the **Verbosity** level.

Note: Some of the build options are not available for some project types.

Item	Description
Pre-Build	Enter the commands in the **Commands:** window that are to be performed before the rest of the build. To display the **Events List** dialog box for creating a command list, click **Edit**.
Pre-Link	For C++ only. Enter the commands in the **Commands:** window that are to be performed before linking. To display the **Events List** dialog box for creating a command list, click **Edit**.
Post-Build	Enter the commands in the **Commands:** window that are to be performed after the build has completed. To display the **Events List**dialog box for creating the command list, click **Edit**.

See Also

Pre-Build Event or Post-Build Event Dialog Box (see page 844)

Creating Build Events (see page 107)

3.2.11.2.12 Compiler

Project ▶ Options ▶ Compiler

Use this dialog box to set the C# compiler options for the current project.

Item	Description
Debug/Release	Indicates the current set of project options. By default, the distributed Debug and Release option sets have settings appropriate for debugging and deploying an application, respectively. The option sets provide an easy way to change project options based on your development activity. To create a user-defined option set, use the **Save as** button.
Save as	Displays a dialog box for naming and saving a user-defined set of project options.
Delete	Deletes the current option set. Only user-defined option sets can be deleted. The distributed Debug and Release option sets can not be deleted.
Optimization	Enables optimizations performed by the compiler to make your output file smaller, faster, and more efficient. This option also tells the common language runtime to optimize code at runtime. Corresponds to /optimize.
Allow unsafe code	Allows code that uses the unsafe keyword to compile. Corresponds to /unsafe.
Treat warnings as errors	Reports all warnings as errors. Any messages that would ordinarily be reported as warnings are instead reported as errors, and the build process is halted (no output files are built). Corresponds to /warnaserror.

Warning level	Specifies the warning level for the compiler to display. **0** Suppresses all warning messages. **1** Displays severe warning messages. **2** Displays level 1 warnings plus certain less severe warnings, such as warnings about hiding class members. **3** Displays level 2 warnings plus certain less severe warnings, such as warnings about expressions that always evaluate to true or false. **4** Displays all level 3 warnings plus informational warnings. Corresponds to /warn. The default is 4.
Debug information	Causes the compiler to generate debugging information and place it in a program database file (`.pdb`) the next time you compile the application. • None — No debugging information will be available. • Full — Enables attaching a debugger to the running program. • PDB only — Allows source code debugging when the program is started in the debugger but will only display assembler when the running program is attached to the debugger. Corresponds to /debug. The default is Full for the Debug configuration, or PDB Only for the Release configuration.
Target platform	Specifies which version of the common language runtime (CLR) can run the assembly. • Any CPU — Compiles your assembly to run on any platform. • x86 — Compiles your assembly to be run by the 32-bit, x86-compatible common language runtime. • x64 — Compiles your assembly to be run by the 64-bit common language runtime on a computer that supports the AMD64 or EM64T instruction set. Corresponds to /platform. The default is x86 for the Debug configuration, or Any CPU for the Release configuration..
Base address	Specifies the preferred base address at which to load the DLL. The default base address for a DLL is set by the .NET Framework common language runtime. This address can be specified as a decimal or hexadecimal number. Corresponds to /baseaddress.
File alignment	Specifies the alignment used for output file sections. Choose a value (in bytes) that specifies the size of sections in the output file. Choices are 512, 1024, 2048, 4096, and 8192 bytes. Corresponds to /filealign. There is no fixed default. If /filealign is not specified, the CLR picks a default at compile time.
Generate XML documentation	Processes documentation comments in the code and creates an XML file named `ProjectDoc.xml` in the same directory as the project file (`.csproj` for MSBuild format). Lines beginning with /// and preceding a user-defined type such as a class, delegate, or interface; a member such as a field, event, property, or method; or a namespace declaration can be processed as comments and placed in the file. Corresponds to /doc.
Do not reference mscorlib.dll	Prevents the import of mscorlib.dll, which defines the entire System namespace. Use this option if you want to define or create your own System namespace and objects. Corresponds to /nostdlib.
Generate overflow checks	Specifies whether an integer arithmetic statement that is not in the scope of the checked or unchecked keywords and that results in a value outside the range of the data type shall cause a run-time exception. Corresponds to /checked.

Codepage	Specifies the codepage to use during compilation if the required page is not the current default codepage for the system. If you compile one or more source code files that were not created to use the default code page on your computer, you can use this option to specify which code page should be used. This option applies to all source code files in your compilation. If the source code files were created with the same codepage that is in effect on your computer or if the source code files were created with UNICODE or UTF-8, you need not use this option. Corresponds to /codepage.
Language version	Specifies whether the compiler will only accept syntax that is included in the ISO/IEC 23270:2003 C# language specification. Corresponds to /langversion. The default is to accept all valid language syntax

Tip: To display the compiler options in the Message

window when you compile a project, choose **Tools ▶ Options ▶ Environment Options** and select the **Show command line** option. The next time you compile a project, the command used to compile the project will be displayed in the **Messages** window.

3.2.11.2.13 Compiler

Project ▶ Options ▶ Compiler

Use this page to set the compiler options for the current project.

Note: Not all of the options described below are available for all types of projects.

Code generation items	Description
Build Configuration	Displays the current named build configuration associated with the options on this page. There are two default build configurations: **Debug** and **Release**. To create additional build configurations, enter a name in this field and click **Save As**. To delete the named build configuration displayed in this field, click **Delete**.
Optimization	Controls code optimization. When enabled (equivalent to {$O+}), the compiler performs a number of code optimizations, such as placing variables in CPU registers, eliminating common subexpressions, and generating induction variables. When disabled, (equivalent to {$O-}), all such optimizations are disabled. Other than for certain debugging situations, you should never have a need to turn optimizations off. All optimizations performed by the Delphi compiler are guaranteed not to alter the meaning of a program. In other words, the compiler performs no "unsafe" optimizations that require special awareness by the programmer. This option can only turn optimization on or off for an entire procedure or function. You can't turn optimization on or off for a single line or group of lines within a routine.
Stack frames	Delphi for Win32 only. Controls the generation of stack frames for procedures and functions. When enabled, (equivalent to {$W+}), stack frames are always generated for procedures and functions, even when they're not needed. When disabled, (equivalent to {$W-}), stack frames are only generated when they're required, as determined by the routine's use of local variables. Some debugging tools require stack frames to be generated for all procedures and functions, but other than that you should never have a need to enable this option.

Pentium-save FDIV	Delphi for Win32 only. Controls generation of floating-point code that guards against the flawed FDIV instruction exhibited by certain early Pentium processors. Windows 95, Windows NT 3.51, and later Windows OS versions contain code that corrects the Pentium FDIV bug system-wide. When enabled (equivalent to {$U+}), all floating-point divisions are performed using a runtime library routine. The first time the floating-point division routine is invoked, it checks whether the processor's FDIV instruction works correctly, and updates the TestFDIV variable (declared in the System unit) accordingly. For subsequent floating-point divide operations, the value stored in TestFDIV is used to determine what action to take. **-1** means that FDIV instruction has been tested and found to be flawed. **0** means that FDIV instruction has not yet been tested. **1** means that FDIV instruction has been tested and found to be correct. For processors that do not exhibit the FDIV flaw, enabling this option results in only a slight performance degradation. For a flawed Pentium processor, floating-point divide operations may take up to three times longer in the enabled state but always produce correct results. In the disabled (equivalent to {$U-}) state, floating-point divide operations are performed using in-line FDIV instructions. This results in optimum speed and code size, but may produce incorrect results on flawed Pentium processors. You should use the disabled state only in cases where you are certain that the code is not running on a flawed Pentium processor.
Record field alignment	Controls alignment of fields in Delphi record types and class structures. If you select option **1** (equivalent to {$A1}) or disable the option (equivalent to {$A-}), fields are never aligned. All record and class structures are packed. If you select **2** (equivalent to {$A2}), fields in record types that are declared without the packed modifier and fields in class structures are aligned on word boundaries. If you select option **4** (equivalent to {$A4}), fields in record types that are declared without the packed modifier and fields in class structures are aligned on double-word boundaries. If you select **8** (equivalent to {$A8} or {$A+}), fields in record types that are declared without the packed modifier and fields in class structures are aligned on quad word boundaries. Regardless of the state of the $A directive, variables and typed constants are always aligned for optimal access. By setting the option to **8**, execution is faster.
Codepage	Enter the codepage for your application's language. Codepage is a decimal number representing a specific character encoding table, and there are standard values for various languages.

Syntax options items	**Description**
Strict var-strings	This option (equivalent to $V directive) is meaningful only for Delphi code that uses short strings, and is provided for backwards compatibility with early versions of Delphi and CodeGear Pascal. The option controls type checking on short strings passed as variable parameters. When enabled (equivalent to {$V+}), strict type checking is performed, requiring the formal and actual parameters to be of identical string types. When disabled (equivalent to {$V-}) (relaxed), any short string type variable is allowed as an actual parameter, even if the declared maximum length is not the same as that of the formal parameter.
Complete boolean eval	Switches between the two different models of Delphi code generation for the AND and OR Boolean operators. When enabled (equivalent to {$B+}), the compiler generates code for complete Boolean expression evaluation. This means that every operand of a Boolean expression built from the AND and OR operators is guaranteed to be evaluated, even when the result of the entire expression is already known. When disabled (equivalent to {$B-}), the compiler generates code for short-circuit Boolean expression evaluation, which means that evaluation stops as soon as the result of the entire expression becomes evident in left to right order of evaluation.

Extended syntax	Provided for backward compatibility. You should not use this option (equivalent to {$X-} mode) when writing Delphi applications. This option enables or disables Delphi's extended syntax: **Function statements.** In the {$X+} mode, function calls can be used as procedure calls; that is, the result of a function call can be discarded, rather than passed to another function or used in an operation or assignment. Generally, the computations performed by a function are represented through its result, so discarding the result makes little sense. Sometimes, however, a function is called because it performs a task such as setting the value of a global variable, without producing a useful result. **The Result variable.** When enabled (equivalent to {$X+}, the predefined variable Result can be used within a function body to hold the function's return value. **Null-terminated strings.** When enabled, Delphi strings can be assigned to zero-based character arrays (array[0..X] of Char), which are compatible with PChar types.
Typed @ operator	Controls the types of pointer values generated by the @ operator and the compatibility of pointer types. When disabled (equivalent to {$T-}), the result of the @ operator is always an untyped pointer (Pointer) that is compatible with all other pointer types. When @ is applied to a variable reference in the enabled (equivalent to {$T+}), the result is a typed pointer that is compatible only with Pointer and with other pointers to the type of the variable. When disabled, distinct pointer types other than Pointer are incompatible (even if they are pointers to the same type). When enabled, pointers to the same type are compatible.
Open parameters	Meaningful only for code compiled supporting huge strings, and is provided for backwards compatibility with early versions of Delphi and CodeGear Pascal. This option, (equivalent to $P) controls the meaning of variable parameters declared using the string keyword in the huge strings disabled (equivalent to {$H-}) state. When disabled (equivalent to {$P-}), variable parameters declared using the string keyword are normal variable parameters, but when enabled (equivalent to {$P+}), they are open string parameters. Regardless of the setting of this option, the openstring identifier can always be used to declare open string parameters.
Huge strings	Delphi for Win32 only. This option (equivalent to the $H directive) controls the meaning of the reserved word string when used alone in a type declaration. The generic type string can represent either a long, dynamically-allocated string (the fundamental type AnsiString) or a short, statically allocated string (the fundamental type ShortString). By default, Delphi defines the generic string type to be the long AnsiString. All components in the component libraries are compiled in this state. If you write components, they should also use long strings, as should any code that receives data from component library string-type properties. The disabled (equivalent to {$H-}) state is mostly useful for using code from versions of Delphi that used short strings by default. You can locally override the meaning of string-type definitions to ensure generation of short strings. You can also change declarations of short string types to string[255] or ShortString, which are unambiguous and independent of the enabled option.
Assignable typed constants	Controls whether typed constants can be modified or not. When enabled (equivalent to {$J+}), typed constants can be modified, and are in essence initialized variables. When disabled (equivalent to {$J-}), typed constants are truly constant, and any attempt to modify a typed constant causes the compiler to report an error. Writable consts refers to the use of a typed const as a variable modifiable at runtime. Old source code that uses writable typed constants must be compiled with this option enabled, but for new applications it is recommended that you use initialized variables and compile your code with the option disabled.

Target platform items	Description (only for .NETprojects)
AnyCPU	The executable runs on any CPU.
x86	The executable runs only on the 32–bit x86 common language runtime.
x64	The executable runs only on the 64–bit common language runtime on computers that support the AMD64 or EM64T instruction set.

Runtime errors items	Description
Range checking	Enables or disables the generation of range-checking code. When enabled, (equivalent to {$R+}), all array and string-indexing expressions are verified as being within the defined bounds, and all assignments to scalar and subrange variables are checked to be within range. If a range check fails, an ERangeError exception is raised (or the program is terminated if exception handling is not enabled). Enabling range checking slows down your program and makes it somewhat larger.
I/O checking	Enables or disables the automatic code generation that checks the result of a call to an I/O procedure. If an I/O procedure returns a nonzero I/O result when this switch is on, an EInOutError exception is raised (or the program is terminated if exception handling is not enabled). When this switch is off, you must check for I/O errors by calling IOResult.
Overflow checking	Controls the generation of overflow checking code. When enabled (equivalent to {$Q+}), certain integer arithmetic operations (+, -, *, Abs, Sqr, Succ, Pred, Inc, and Dec) are checked for overflow. The code for each of these integer arithmetic operations is followed by additional code that verifies that the result is within the supported range. If an overflow check fails, an EIntOverflow exception is raised (or the program is terminated if exception handling is not enabled). This switch is usually used in conjunction with the range checking option ($R switch), which enables and disables the generation of range-checking code. Enabling overflow checking slows down your program and makes it somewhat larger.

Debugging items	Description
Debug information	Enables or disables the generation of debug information. This information consists of a line-number table for each procedure, which maps object-code addresses into source text line numbers. For units, the debug information is recorded in the unit file along with the unit's object code. Debug information increases the size of unit file and takes up additional memory when compiling programs that use the unit, but it does not affect the size or speed of the executable program. When a program or unit is compiled with this option enabled (equivalent to {$D+}), the integrated debugger lets you single-step and set breakpoints in that module. The **Include debug info** and **Map file** options (on the **Linker** page of the **Project Options** dialog) produce complete line information for a given module only if you've compiled that module with this option set on. This option is usually used in conjunction with the **Local symbols** option (the $L switch), which enables and disables the generation of local symbol information for debugging.
Local symbols	Enables or disables the generation of local symbol information. Local symbol information consists of the names and types of all local variables and constants in a module, that is, the symbols in the module's implementation part and the symbols within the module's procedures and functions. For units, the local symbol information is recorded in the unit file along with the unit's object code. Local symbol information increases the size of unit files and takes up additional memory when compiling programs that use the unit, but it does not affect the size or speed of the executable program. When a program or unit is compiled with this option enabled (equivalent to {$L+}), the integrated debugger lets you examine and modify the module's local variables. Furthermore, calls to the module's procedures and functions can be examined by way of the View | Call Stack. The Include TD32 debug info and Map file options (on the **Linker** page of the **Project Options** dialog) produce local symbol information for a given module only if that module was compiled with this option set on. This option is usually used in conjunction with the **Debug information** option, which enables and disables the generation of line-number tables for debugging. This option is ignored if the compiler has the **Debug information** option disabled.
Reference info	Generates symbol reference information used by the **Code Editor** and the **Project Manager**. Corresponds to {$Y}. If **Reference info** and **Definitions only** are both checked ({$YD}), the compiler records information about where identifiers are defined. If **Reference info** is checked but **Definitions only** is unchecked ({$Y+}), the compiler records information about where each identifier is defined and where it is used. These options have no effect unless **Debug information** and **Local symbols** (see above) are selected.
Definitions only	See description of **Reference info**.

Assertions	Enables or disables the generation of code for assertions in a Delphi source file. The option is enabled (equivalent to {$C+}) by default. Since assertions are not usually used at runtime in shipping versions of a product, compiler directives that disable the generation of code for assertions are provided. Uncheck this option to disable assertions.
Use Debug DCUIL/DCUs	The debug DCUILs (.NET) or DCUs (Win32) contain debug information and are built with stack frames. When this option is checked, the compiler prepends the debug DCUIL/DCU path to the unit search path specified in **Debug Source Path** on the **Directories/Conditionals** page.

Documentation items	Description
Generate XML Documentation	Generates a file containing the XML representation in your project directory. Corresponds to the --doc compiler switch.

General item	Description
Default	Saves the current settings as the default for each new project.

Tip: To display the compiler options in the Messages

window when you compile a project, choose **Tools▶Options▶Environment Options** and select \ **Show command line**. The next time you compile a project, both the command used to compile the project and the response file are displayed in the **Messages** window. The response file lists the compiler options and the files to be compiled.

See Also

Compiling (see page 2)

Creating Named Build Configurations for C++ (see page 107)

Creating Named Build Configurations in Delphi (see page 108)

3.2.11.2.14 Compiler Messages

Project▶Options▶Compiler Messages

Use this page to control the information that the compiler provides at compile time.

Item	Description
Build Configuration	Displays the current named build configuration associated with the options on this page. There are two default build configurations: **Debug** and **Release**. To create additional build configurations, enter a name in this field and click **Save As**. To delete the named build configuration displayed in this field, click **Delete**.
Show hints	Enables hints during compile time.
Show warnings	Enables warnings during compile time.
Warnings	Allows you to select which warnings are displayed during compile time.
Default	Saves the current settings as the default for each new project.

See Also

Compiler Errors (see page 311)

3.2.11.2.15 Compiler (Visual Basic)

Project ▶Options▶Compiler

Use this dialog box to set the Visual Basic compiler options for the current project. For additional information about the Visual Basic compiler options, see the .NET Framework SDK online Help.

Item	Description
Debug/Release	Indicates the current set of project options. By default, the distributed Debug and Release option sets have settings appropriate for debugging and deploying an application, respectively. The option sets provide an easy way to change project options based on your development activity. To create a user-defined option set, use the **Save as** button.
Save as	Displays a dialog box for naming and saving a user-defined set of project options.
Delete	Deletes the current option set. Only user-defined option sets can be deleted. The distributed Debug and Release option sets can not be deleted.
Debug information	Causes the compiler to generate debugging information and place it in a program database file (.pdb) the next time you run the application. Corresponds to /debug.
Optimization	Enables optimizations performed by the compiler to make your output file smaller, faster, and more efficient. This option also tells the common language runtime to optimize code at runtime. Corresponds to /optimize.
Treat warnings as errors	Treats all warnings as errors. Any messages that would ordinarily be reported as warnings are instead reported as errors, and the build process is halted (no output files are built). Corresponds to /warnaserror.
Show warnings	Generates compiler warnings. Corresponds to /nowarn.
Conditional Defines	Enter symbols referenced in conditional compiler directives. Separate multiple symbols with semicolons.
Output Directory	Specifies where the compiler will put the executable file. Corresponds to /out.

Tip: To display the compiler options in the Message

window when you compile a project, choose **Tools▶Options▶Environment Options** and select the **Show command line** option. The next time you compile a project, the command used to compile the project and the response file will be displayed in the **Messages** window. The response file lists the compiler options and the files to be compiled.

3.2.11.2.16 Components

Project ▶Options▶Packages▶Components button

Use this dialog box to display the components in the selected package, along with the icons that represent the components in the **Tool Palette** if the package is installed.

3.2.11.2.17 Debugger

Project▶Options▶Debugger

Use this dialog to pass command-line parameters to your application, specify a host executable for testing a DLL, or load an executable into the debugger. This dialog is also available from **Run ▶ Parameters**.

Item	Description
Host Application	Specifies the path to an executable file. (Click **Browse** to bring up a file-selection dialog.) If the current project is a DLL or package, use this edit box to specify a host application that calls it. You can also enter the name of any executable that you want to run in the debugger. If you want to run the project that you have open, there is no need to enter anything in **Host Application**.
Parameters	Specifies the command-line parameters to pass to your application. Enter the command-line arguments you want to pass to your application (or the host application) when it starts. You can use the drop-down button to choose from a history of previously specified parameters.
Working Directory	Specifies name of the directory to use for the debugging process. By default, this is the same directory as the one containing the executable of your application.
Source Path	Specifies the directories containing the source files. By default, the debugger searches paths defined by the compiler. If the directory structure has changed since the last compile or if the debugger cannot find a source file, a path can be entered here to include the file in the debugging session. Click the ellipsis button to display a dialog box that allows you to edit an ordered list of directory source paths Additional directories are searched in the following order: 1. **Debug Source path** (this option). 2. The **Browsing path**, as specified for Delphi for Win32 or .NET or for C++: — For Delphi for Win32, the **Tools ▶ Options ▶ Environment Options ▶ Delphi Options ▶ Library — Win32** page. — For Delphi.NET, the **Tools ▶ Options ▶ Environment Options ▶ Delphi Options ▶ Library — NET** page. — For C++,**Tools ▶ Options ▶ Debugger Options ▶ Borland Debuggers** page. 3. **Debug Source path**, specified on the **Tools ▶ Options ▶ Debugger Options ▶ Borland Debuggers** page. **Note:** The **Browsing path** is not used for C# or Visual Basic.
Default	Saves the current settings as the default for each new project.

3.2.11.2.18 Description

Project ▶ Options ▶ Description

Use the **Description** page to specify a description for the package, the uses of the package, and how the package is built.

Note: This page appears only if you are developing a package.

Description option	Description
Description	Specifies the description that appears when the package is installed.

Usage options	Description
Designtime only	The package is installable on the **Tools Palette**.
Runtime only	The package can be deployed with an application.
Designtime and runtime	The package is both installable and deployable.

Build control options	Description
Rebuild as needed	Builds the package as needed.
Explicit rebuild	Builds the package only when you choose Project->Build. Use this option if the package is low-level and does not change often.

Package name options	Description
LIB Prefix	Adds a specified prefix to the output file name.
LIB Version	Adds a second extension to the output file name after the `.bpl` extension. For example, specify `2.1.3` here for `Package1` to generate `Package1.bpl.2.1.3`.
LIB Suffix	Adds a specified suffix to the output file name before the extension. For example, specify `-2.1.3` here for `Package1` to generate `Package1-2.1.3.bpl`.

General option	Description
Default	Saves the current settings as the default for each new project.

3.2.11.2.19 Directories/Conditionals

Project ▶ Options ▶ Directories/Conditionals

Use this page to set your directory and conditional defines paths for C# and VB.

Item	Description
Debug/Release	Indicates the current set of project options. By default, the distributed Debug and Release option sets have settings appropriate for debugging and deploying an application, respectively. The option sets provide an easy way to change project options based on your development activity. To create a user-defined option set, use the **Save as** button.
Save as	Displays a dialog box for naming and saving a user-defined set of project options.
Delete	Deletes the current option set. Only user-defined option sets can be deleted. The distributed Debug and Release option sets cannot be deleted.
Conditional Defines	Enter symbols referenced in conditional compiler directives. Separate multiple symbols with semicolons. Corresponds to the /define compiler switch for both C# and VB.
Output directory	Specifies where the compiler should put the executable file. Corresponds to the /out compiler switch for both C# and VB.
Default	Saves the current settings as the default for each new project.

3.2.11.2.20 Directories/Conditionals

Project ▶ Options ▶ Directories/Conditionals

Use this page to set your directory and conditional defines paths. You can also set the namespace prefix, to simplify your `uses` clause.

Item	Description
Build Configuration	Displays the current named build configuration associated with the options on this page. There are two default build configurations: **Debug** and **Release**. To create additional build configurations, enter a name in this field and click **Save As**. To delete the named build configuration displayed in this field, click **Delete**.
Output directory	Specifies where the compiler should put the executable file.
Unit output directory	Specifies a separate directory to contain the `.dcu` (Win32) or `.dcuil` (.NET) files.
Search path	Specifies the location of your source files. Only those files on the compiler's search path or the library search path are included in the build. If you try to build your project with a file not on the search path, you will receive a compiler error. You must include the entire search path. Separate multiple directory path names with a semicolons. Whitespace before or after the semicolon is allowed but not required. Relative and absolute path names are allowed, including path names relative to the current position. If you check the **Use Debug DCUILs** option on the **Project**▶**Options**▶**Compiler** page, the **Debug DCUIL path** from **Tools**▶**Debugger Options**▶**CodeGear .NET Debugger** is prepended to this search path.
Package output directory	Specifies where the compiler puts generated package files.
DCP/DCPIL output directory	Specifies where the `.dcp` (Win32) or `.dcpil` (.NET) file is placed at compilation time. If left blank, the global **DCP/DCPIL output directory** specified in the **Tools**▶**Options**▶**Environment Options**▶**Delphi Options**▶**Library** page is used instead.
Conditional defines	Specify the symbols referenced in conditional compiler directives. Separate multiple defines with semicolons.
Unit aliases	Useful for backwards compatibility. Specify alias names for units that may have changed names or were merged into a single unit. The format is `<oldunit>=<newunit>` (for example, `Forms=Xforms`). Separate multiple aliases with semicolons. The default value for Delphi is `WinTypes=Windows;WinProcs=Windows.Default`.
Namespace prefixes	Specifies the prefixes for namespaces, to allow you to create a shorthand version of the namespace in the uses clause in your code file. For example, instead of writing `Borland.Vcl.DB`, you could specify `Borland.Vcl` as your namespace prefix. In the uses clause, you could then specify `uses DB;.`
Default Namespace	Indicates the default namespace used for all units in the project.
Default	Saves the current settings as the default for each new project.

3.2.11.2.21 Debugger Environment Block

Project▶**Options**▶**Debugger**▶**Environment Block**

Use this page to indicate which environment variables are passed to your application while you are debugging it. This page is also available by choosing **Run**▶**Parameters**.

Item	Description
System variables	Lists all environment variables and their values defined at a system level. You cannot delete an existing system variable, but you can override it.
Add Override...	Displays the **Override System Variable** dialog box, allowing you to modify an existing system variable to create a new user override. This button is disabled until you select a variable in the **System variables** list.
User overrides	Lists all defined user overrides and their values. A user override takes precedence over an existing system variable until you delete the user override.

New	Displays the **New User Variable** dialog box allowing you to create new user override to a system variable.
Edit	Displays the **Edit User Variable** dialog box allowing you to change the user override currently selected in the **User overrides** list.
Delete	Deletes the user override currently selected in the **User overrides** list.
Include System Variables	Passes the system environment variables to the application you are debugging. If unchecked, only the user overrides are passed to your application.
Default	Saves the current settings as the default for each new project.

3.2.11.2.22 Forms

Project ▶ Options ▶ Forms

Use this page to select the main form for the current VCL Forms project and to choose which of the available forms are automatically created when your application begins.

Item	Description
Main form:	Indicates which form users see when they start your application. Use the drop-down list to select the main form for the project. The main form is the first form listed in the **Auto-create forms** list box.
Auto-create forms:	Lists the forms that are automatically added to the startup code of the project file and created at runtime. These forms are automatically created and displayed when you first run your application. You can rearrange the create order of forms by dragging and dropping forms to a new location. To select multiple forms, hold down the `SHIFT` key while selecting the form names.
Available forms:	Lists those forms that are used by your application but are not automatically created. If you want to create an instance of one of these forms, you must call its Create method.
Arrow buttons	Use the arrow buttons to move files from one list box to the other.
Default	Saves the current settings as the default for each new project.

3.2.11.2.23 Linker

Project ▶ Options ▶ Linker

Use this page to set linker options for your application.

Note: Not all of the options described below are available for all types of projects.

Build Configuration options	Description
Build Configuration	Displays the current named build configuration associated with the options on this page. There are two default build configurations: **Debug** and **Release**. To create additional build configurations, enter a name in this field and click **Save As...**. To delete the named build configuration displayed in this field, click **Delete**.
Save As...	Saves the current configuration to a build configuration with the name specified in the **Build Configuration** field.
Delete	Deletes the configuration specified in the **Build Configuration** field..

Map file items	Description
Off	The linker does not produce a map file.
Segments	Causes the linker to produce a map file that includes a list of segments, the program start address, and any warning or error messages produced during the link.
Publics	Causes the linker to produce a map file that includes a list of segments, the program start address, any warning or error messages produced during the link, and a list of alphabetically sorted public symbols.
Detailed	Causes the linker to produce a map file that includes a list of segments, the program start address, any warning or error messages produced during the link, a list of alphabetically sorted public symbols, and an additional detailed segment map. The detailed segment map includes the segment address, length in bytes, segment name, group, and module information.

Linker output items	Description
Generate DCUs	Creates the standard Delphi `.dcu` format files
Generate C object files	Creates a C object file for linking with a C program (no name mangling).
Generate C++ object files	Creates a C++ object file for linking with C++ (uses C++ name mangling).
Include namespaces	Include C++ namespace information in the OBJ and HPP files generated.
Export all symbols	Include symbol information in the OBJ and HPP files generated.
Generate header files	Include header file information in the OBJ and HPP files generated.
Generate all C++ Builder files	Select this to include all namespace, symbol, and header file information in the package. This option applies only to packages. It is recommended that you check this item rather than the items under **Generate C++ object files**.

EXE and DLL options items	Description
Generate console application	Causes the linker to set a flag in the application's .exe file indicating a console mode application.
Generate .PDB debug info file (.NET) Include TD32 debug info (Win32)	Causes the compiler to generate debugging information and place it in a program database file the next time you run the application.
Include remote debug symbols	Check this if you are using remote debugging.
Generate .DRCIL file	Creates a `.drcil` file containing string resources, which can be compiled into a resource file.

Memory sizes items	Description
Min stack size	Indicates the initial committed size of the stack (only applicable to executable projects; disabled for DLLs). Memory-size settings can also be specified in your source code with the $M compiler directive.

Max stack size	Indicates the total reserved size of the stack (only applicable to executable projects; disabled for DLLs). Memory-size settings can also be specified in your source code with the $M compiler directive.
Image base	Specifies the preferred load address of the compiled image. This value is typically only changed when compiling DLLs.

Description item	Description
EXE Description	This field can contain a string of up to 255 characters. The string is linked to $D and included in the executable file. It is most often used to insert copyright information into the application. Copyright information can also be included as part of the VersionInfo file. Note that this option is only applicable to DLLs and application executables but not for packages.

COM items	Description
Auto Register Type Library	Available only for projects with a type library. Adds an entry for the type library to your system's Windows registry.
Generate Import Assembly	Available only for projects with a type library. After the project is built, `tlbimp.exe` is executed to generate an interop assembly.

General item	Description
Default	Saves the current settings as the default for each new project.

3.2.11.2.24 Project Options

Project ▶ Options

Use this dialog box to set project options.

In the left pane, click an item in a category to display its option page. If you leave the cursor over text describing an option, a tool tip gives you an option description, its default value, and a switch for the option if one exists.

Each configuration, except the **Base**, is based on another configuration. In the **Project Manager** pane, the **Build Configurations** node under the project represents the **Base** configuration. All configurations are listed under the **Build Configurations** node in a hierarchical list that shows the parent-child relationship for each configuration.

If an option's value differs from its parent's configuration, its associated text is boldface. To revert to the parent configuration value, right-click the option text and click **Revert** on the context menu. If you change option values, you can save your set of changes in a new configuration or a named option set. You can switch to this configuration or load this option set into the active configuration.

Build Configuration options	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration. The **Build Configuration** drop-down menu lists the configurations for that project—not option sets. Each project has its own list of configurations, independent of other projects. An active configuration's option values can be saved to an option set file with the **Save As...** button. An option set may be applied entirely or partially to a project's active configuration with the **Load...** button. Three initial configurations are provided: Base, Debug, and Release. The Base configuration provides a base set of configuration options. It cannot be deleted, so you always have at least one configuration. Some options that contain a list of items, such as defines or paths, have a **Merge** check box. If checked, the IDE merges the option's list with that of its immediate ancestor's configuration's list for that option. Note that the IDE does not actually change the contents of the option, but acts as if the list included the ancestor's list. If the ancestor's **Merge** check box is also checked, the IDE also merges this ancestor's list for that option, and so on up the inheritance chain. If unchecked, the IDE uses only the items in the current configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set. Only the values in the current active configuration that are different than their parent's configuration are saved in the option set file. This includes any values that you changed in the active configuration as well as any that were already changed from their default values in the configuration.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration. You have three choices for how you load the values: **Overwrite**, **Replace**, or **Preserve**.

General option	Description
Default	Saves the current settings as the default for each new project.

See Also

Apply Option Set dialog (↗ see page 828)

Build Configurations Overview (↗ see page 5)

Build Configuration Manager (↗ see page 828)

Working With Named Option Sets (↗ see page 113)

Applying the Active Build Configuration (↗ see page 104)

3.2.11.2.25 Packages

Project ▶ Options ▶ Packages

Use this page to specify the design time packages installed in the IDE and the runtime packages required by your project.

Item	Description
Design packages	Lists the design time packages available to the IDE and all projects. Check the design packages you want to make available for your current project.
Add	Installs a design time package. The package will be available to your current project.

Remove	Deletes the selected package. The package becomes unavailable for your current project.
Edit	Opens the selected package in the Package Editor if the source code or .dcp file is available.
Components	Displays a list of the components included in the selected package.
Runtime packages	Determines which runtime packages to use when the executable file is created. Separate package names with semicolons. As packages are installed and uninstalled, the runtime package list is updated. The product automatically adds runtime packages that are required by installed design time packages.
Build with runtime packages	Dynamically links the runtime packages in your project and enables the runtime packages edit box.
Ellipsis	Displays **Runtime Package** dialog box, allowing you to specify the name of a runtime package to add to the **Runtime packages** list.
Default	Saves the current settings as the default for each new project.

3.2.11.2.26 Pre-Build, Pre-Link, or Post-Build Events

Use this dialog box to create a list of commands and macros to execute at certain points in the build process. Enter any valid list of DOS commands. The commands and their results are displayed in the on the **Output** tab of the **Messages** pane.

Project ▶ Options ▶ Build Events ▶ Edit...

Note: Pre-Link events are available only for C++ projects.

Item	Description
Execute when	Executes the specified events always or only if the target is out of date. This option applies only to Post-Build events.
Cancel build on error	Cancels the project build if a command returns a nonzero error code.
Commands	Specifies the commands to execute. Separate each command by a newline character.
Macros	Lists macros available to use as command arguments. Clicking a macro places its text in the **Commands** window at the cursor position.

3.2.11.2.27 Signing

Project ▶ Options ▶ Signing

Use the Signing dialog to sign the assembly produced by the project using the Microsoft Strong Name (SN) utility.

Note: Signing can only be used in the .NET framework.

Item	Description
Sign the assembly	Use the Microsoft Strong Name utility to sign the assembly (executable, DLL, etc.) produced by the project. Refer to the Microsoft .NET SDK for more information about signing.
Strong name key file	Specifies the file with the signature. Click [...] to display a dialog to browse for the file.
Delay sign only	Delay signing the assembly produced by the project. You would use this only if there is some post build step you need to perform on the assembly. This is typically not checked, since failing to sign prevents you from running or debugging the assembly.

Warning: If you sign the assembly, delaying signing prevents you from running or debugging the assembly produced by the project.

3.2.11.2.28 Debugger Symbol Tables

Project ▶ Options ▶ Debugger ▶ Symbol Tables

Use this dialog box to specify the location of the symbols tables to be used during debugging. This dialog is also available from **Run ▶ Parameters**.

Item	Description
Debug symbols search path:	Specifies the directory containing the symbol tables used for debugging. This path is used if you check the **Load all symbols** check box.
Load all symbols	Sets the state of the **Mappings from Module Name to Symbol Table Path** list. If checked, the list is disabled and all symbol tables are loaded by the debugger. The debugger uses the **Debug symbols search path** to search for the symbol table file associated with each module loaded by the process being debugged. If unchecked, the **Mappings from Module Name to Symbol Table Path**list is enabled and its settings are used.
Mappings from Module Name to Symbol Table Path	Displays the current mapping of each module name to a symbol table search path that is defined for the project. Use the up and down arrows (to the right of the dialog) to move the selected item up or down in the list. The debugger searches this list, in order, to find a match for the name of the module being loaded. When the debugger finds a matching module name, it uses the corresponding path to locate that module's symbol table. For example, if module **foo123.dll** is loaded, and the list shows **foo*.dll** as the first item and ***123.dll** as a later item, the debugger only uses the symbol table path for **foo*.dll**, even though both items match the module being loaded.
Load symbols for unspecified modules	Specifies whether symbol tables for modules not in the **Mappings from Module Name to Symbol Table Path** list (either explicitly or via a file mask) are loaded during debugging. If checked, the symbol tables for modules not specified are loaded using the **Debug symbols search path**. If unchecked, symbol tables are loaded only for modules in the list.
New	Displays the **Add Symbol Table Search Path** dialog, where you can specify a module name and an associated search table path. The module and path are added to the **Mappings from Module Name to Symbol Table Path** list. Note that you can add a blank path to prevent a symbol table for a module from being loaded.
Edit	Displays the selected module and path in the **Add Symbol Table Search Path** dialog, enabling you to edit the module name or path that displays in the **Mappings from Module Name to Symbol Table Path** list.
Delete	Removes the selected module from the **Mappings from Module Name to Symbol Table Path** list.
Default	Saves the current settings as the default for each new project.

See Also

Debugging Applications (see page 10)

Preparing Files for Remote Debugging (see page 128)

3.2.11.2.29 Version Info

Project ▶ Options ▶ Version Info

Use this dialog box to specify version information for a Delphi Win32 project. When version information is included, a user can right-click the program icon and select properties to display the version information.

Item	Description
Include version information in project	Determines whether the user can view product identification information. For this option to be available in console applications, you must add {$R *.res} to your project source.
Module version number	Sets hierarchical nested version, release, and build identification. Major, Minor, Release, and Build each specify an unsigned integer between 0 and 65,535. The combined string defines a version number for the application, for example 2.1.3.5. Check **Auto-increment build number** to increment the build number each time the **Project ▶ Build <Project>** menu is selected. Other compilations do not change the build number.
Module attributes	Indicates the intent of this version: whether for debugging, pre-release, or other purposes. Module attributes can be included in the version information and are for informational use only. If a project is compiled in debug mode, the debug flag is included in the version information. You can select each of the remaining flags as needed. **Debug build** Indicates that the project was compiled in debug mode. **Pre-release** Indicates the version is not the commercially released product. **DLL** Indicates that the project includes a dynamic-link library. **Special build** Indicates that the version is a variation of the standard release. **Private build** Indicates that the version was not built using standard release procedures.
Language	Indicates which Code Page the user's system requires to run the application. You can only choose a language that is listed in the **Control Panel Regional Settings** dialog of your computer. Some versions of the Windows operating system do not include support for all languages (such as Far Eastern languages), and you may need to install the appropriate Language Pack before you can use those languages.
Key/Value list box	Sets typical product identification properties. Key entries can be edited by selecting the key and reentering the name. Key entries can be added by right-clicking within the Key/Value table and selecting **Add Key**.
Default	Saves the current settings as the default for each new project.

Tip: To obtain version information programmatically from your compiled Win32 application, use the Windows GetFileVersionInfo and VerQueryValue API functions.

3.2.11.3 COM Imports

Project ▶ Add Reference

Adds a COM type library to the current project.

The **Add Reference** dialog box is also available in the **Project Manager** by right-clicking a **References** folder and choosing Add Reference.

Item	Description
TypeLib Name	The name of the type library.
TypeLib Version	The version of the type library.
TypeLib Path	The location of the type library.
Add Reference	Adds the selected (highlighted) reference to the **New References** list.
Browse	Displays a dialog box allowing you to navigate to a type library.
Remove	Removes the reference currently selected in the **New References** list from the list.
OK	If the **New References** list contains any references, they are added to the project when you click **OK**.

Tip: Click any column heading to sort the display.

3.2.11.4 C++ Project Options

Topics

Name	Description
ATL (↗ see page 847)	**Project ▶ Options ▶ ATL** Use this dialog box to specify ATL options. These options apply to every COM server in the project.
C++ Compiler (↗ see page 848)	
Folder or Directory View (↗ see page 867)	Use this dialog box to add a folder node to the active project. You can use a folder node or a directory view to browse frequently used files that are not part of your project.
Resource Compiler (↗ see page 867)	
DCC32 (↗ see page 869)	
Find Option (↗ see page 878)	**Project ▶ Options** Use this dialog box to find a specific option for the selected build tool.
iLink 32 (↗ see page 878)	
List Editor (↗ see page 885)	**Project ▶ Options ▶ various paths** Use this dialog box to edit a list of semicolon-delimited strings. **Note:** Not all of the options described below are available for all types of projects.
Implib (↗ see page 886)	
Paths and Defines (↗ see page 887)	**Project ▶ Options ▶ Paths and Defines** Use this dialog box to set project paths and defines.
Project Properties (↗ see page 888)	**Project or Tools ▶ Options ▶ Project Properties or Environment Options\C++ Options** Use this dialog box to set Project Properties that control certain aspects of how the project is managed in the IDE. Note that this dialog can be displayed from either **Project Options** or **Tools Options**. If options are set in the dialog from **Project Options**, they apply only to that project. If options are set in the dialog from **Tools Options**, they apply to new projects.
tasm32 (↗ see page 889)	
Unavailable Options (↗ see page 893)	**Project ▶ Options** Some project options are no longer available in C++ Builder 2007. They may be available by using the tool switches. For reference, they are listed here by major topics: C++ Compiler, Resource Compiler. Pascal Compiler, IDL to C++ Compiler, Linker, Librarian, and Turbo Assembler.

3.2.11.4.1 ATL

Project ▶ Options ▶ ATL

Use this dialog box to specify ATL options. These options apply to every COM server in the project.

Item	Description
Single Use	An instance of the COM server object is created for each client.
Multiple Use	All clients operate on a single instance of the COM server object.
APARTMENTTHREADED	An object is referenced only by the thread in which it was constructed. Use this option for projects that contain only single-threaded and apartment-threaded objects.
MULTITHREADED	Objects can be referenced by any thread. Use this option for projects that contain free-threaded, both-threaded, or neutral-threaded objects.

Single	All COM server objects are implemented using a single thread.
Apartment	The project may be multi-threaded, but each instance of the COM server object must has its own dedicated thread for OLE calls.
Free	The project may be multithreaded and each instance of the COM server object can receive simultaneous requests from multiple threads. Your code must provide thread concurrency support.
Both	Same as **Free** except that outgoing calls such as callbacks are guaranteed to execute in the same thread.
Neutral	Multiple clients can call the object on different threads at the same time, but COM ensures that no two calls conflict.
Trace Query Interface	Sends a message to the event log whenever a client makes a `QueryInterface` call. The event log also shows the status of the call.
Check Ref Counts	message is sent to the event log whenever the reference count of a COM server object is increased or decreased. When **Check Ref Counts** is enabled, an assertion occurs if the project attempts to release the object from memory with a nonzero reference count.
General Tracing	Sends a messages to the event log when an ATL function is called.

Note: Threading Model

options are provided only for backward compatibility. You now specify threading models on a per-object basis.

See Also

ATL Overview (MSDN)

3.2.11.4.2 C++ Compiler

Topics

Name	Description
C++ Compiler Advanced Compilation (↗ see page 849)	**Project ▶ Options ▶ C++ Compiler ▶ Advanced Compilation** Use this dialog box to set C++ Compiler General Compilation options.
C++ Compiler C++ Compatibility (↗ see page 851)	**Project ▶ Options ▶ C++ Compiler ▶ C++ Compatibility** Use this dialog box to set specifically C++ Compiler Compatibility options. These options provide backward compatibility with previous versions of the compiler. In general, these options should not be set to true unless such compatibility is required. Their default value is false.
C++ Compiler C++ Compilation (↗ see page 853)	**Project ▶ Options ▶ C++ Compiler ▶ C++ Compilation** Use this dialog box to set C++ Compiler options.
C++ Compiler Debugging (↗ see page 856)	**Project ▶ Options ▶ C++ Compiler ▶ Debugging** Use this dialog box to set C++ compiler debugging and CodeGuard options.
C++ Compiler General Compatibility (↗ see page 857)	**Project ▶ Options ▶ C++ Compiler ▶ General Compatibility** Use this dialog box to set C++ Compiler General Compatibility options. Some of these options provide backward compatibility with previous versions of the compiler, and their default value is false. In general, such options should not be set to true unless such compatibility is required.
C++ Compiler General Compilation (↗ see page 858)	**Project ▶ Options ▶ C++ Compiler ▶ General Compilation** Use this dialog box to set C++ Compiler General Compilation options.
C++ Compiler (↗ see page 861)	**Project ▶ Options ▶ C++ Compiler** This is the top-level node of the C++ Compiler command line options. **Note:** Options marked with an asterisk (*) on the options pages are the default values.
C++ Compiler Optimizations (↗ see page 862)	**Project ▶ Options ▶ C++ Compiler ▶ Optimizations** Use this dialog box to set C++ Compiler Optimization options.

C++ Compiler Output (↗ see page 863)	**Project ▶ Options ▶ C++ Compiler ▶ Output** Use this dialog box to set C++ Compiler Output options.
C++ Compiler Paths And Defines (↗ see page 864)	**Project ▶ Options ▶ C++ Compiler ▶ Paths and Defines** Use this dialog box to set C++ Compiler Paths and Defines options.
C++ Compiler Precompiled Headers (↗ see page 865)	**Project ▶ Options ▶ C++ Compiler ▶ Precompiled headers** Use this dialog box to set C++ Compiler Precompiled headers options.
C++ Compiler Warnings (↗ see page 866)	**Project ▶ Options ▶ C++ Compiler ▶ Warnings** Use this dialog box to set C++ Compiler Warning options.

3.2.11.4.2.1 C++ Compiler Advanced Compilation

Project ▶ Options ▶ C++ Compiler ▶ Advanced Compilation

Use this dialog box to set C++ Compiler General Compilation options.

Build Configuration options	Description
Build Configuration	Displays the active build configuration. You can use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

Floating point options	Description
Fast floating point (-ff)	Floating-point operations are optimized without regard to explicit or implicit type conversions. Calculations can be faster than under ANSI operating mode. The purpose of the fast floating-point option is to allow certain optimizations that are technically contrary to correct C semantics. `double x; x = (float) (3.5*x);` To execute this correctly, `x` is multiplied by 3.5 to give a double that is truncated to float precision, then stores as a double in x. Under fast floating-point operation, the long double product is converted directly to a double. Since very few programs depend on the loss of precision on passing to a narrower floating-point type, fast floating point is on by default. When this option is disabled (-ff-), the compiler follows strict ANSI rules regarding floating-point conversions. (Default = true)
Correct FDIV flaw (-fp)	Some early Pentium chips do not perform specific floating-point division calculations with full precision. Although chances of encountering this problem are slim, this switch inserts code that emulates floating-point division, so that you are assured of the correct result. This option decreases your program's FDIV instruction performance. Use of this option only corrects FDIV instructions in modules that you compile. The runtime library also contains FDIV instructions that are not modified by setting this switch. To correct the runtime libraries, you must recompile them using this switch. The following functions use FDIV instructions in assembly language that are not corrected if you use this option: `acos`, `acosl`, `acos`, `asinasinl`, `atanatan2`, `atan2latanl`, `coscosh`, `coshlcosl`, `expexpl`, `fmodfmodl`, `powpow10`, `pow10lpowl`, `sinsinh`, `sinhlsinl`, `tantanh`, `tanhltanl` In addition, this switch does not correct functions that convert a floating-point number to or from a string (such as **printf** or **scanf**). (Default = false)

Quiet floating point compares (-fq)	Use the quiet floating point instruction (FUCOMP). (Default = true)

Strings options	Description
Writable strings (-dw)	Put memory allocated for strings into the writable data segment. (Default = false)
Read-only strings (-dc)	Put memory allocated for strings into the read-only data segment. (Default = false)
Merge duplicate strings (-d)	Merges two literal strings when one matches another. This produces smaller programs (at the expense of a slightly longer compile time), but can introduce errors if you modify one string. (Default = false)

Other options	Description
Code page (-CP)	Enables support for user-defined code pages. Its primary use is to tell the compiler how to parse and convert multi-byte character strings (MBCS). There are two distinct areas where code pages come into effect: 1. String constants, comments, #error, and #pragma directives 2. Wide-char string constants (as specified by L'<MBCS string>') For MBCS strings belonging to the first set, you must specify the correct codepage using a call to the Windows API function **IsDBCSLeadByteEx**. Using this function, specify the code page to correctly parse the MBCS strings for a particular locale (this, for example, enables the compiler to correctly parse backslashes in MBCS trail bytes). For MBCS strings belonging to the second set (wide-char string constants), specify the correct code page to convert the MBCS strings to Unicode strings using the Windows API function **MultiByteToWideChar**. **Syntax** Enable code paging with the following command-line switch: `-CPnnnn` In this syntax, *nnnn* is the decimal value of the code page you need to use for your specific locale. The following rules apply: 1. When setting code paging, numeric settings for nnnn must adhere to the Microsoft NLS Code Page ID values. For example, use 437 for United States MS-DOS applications. Use 932 for Japanese. 2. The numeric value *nnnn* must be a valid code page supported by the OS. 3. Users may need to install the relevant Windows NLS files to make certain Asian locales and code pages accessible. Refer to the Microsoft NLS Code Page documentation for specifics. 4. If you do not specify a code page value, the compiler calls the Windows API function **GetACP** to retrieve the system's default code page and uses this value when handling strings as indicated above. The default is to not use a code page.
Other options	Any additional options to pass to the compiler.
Default char to unsigned (-K)	The compiler treats **char** declarations as if they were **unsigned char** type, which provides compatibility with other compilers. (Default = false)

Source options	Description
Identifier length (-i)	Specifies the number of significant characters (those which are recognized by the compiler) in an identifier. Except in C++, which recognizes identifiers of unlimited length, all identifiers are treated as distinct only if their significant characters are distinct. This includes variables, preprocessor macro names, and structure member names. Valid numbers for length are 0, and 8 to 250, where 0 means use the maximum identifier length of 250. By default, C++Builder uses 250 characters per identifier. Other systems (including some UNIX compilers) ignore characters beyond the first eight. If you are porting to other environments, you might want to compile your code with a smaller number of significant characters, which helps you locate name conflicts in long identifiers that have been truncated.
Enable nested comments (-C)	Nests comments in your C and C++ source files. Nested comments are not allowed in standard C implementations, and they are not portable. (Default = false)

General option	Description
Default	Saves the current settings as the default for each new project.

3.2.11.4.2.2 C++ Compiler C++ Compatibility

Project ▶ Options ▶ C++ Compiler ▶ C++ Compatibility

Use this dialog box to set specifically C++ Compiler Compatibility options.

These options provide backward compatibility with previous versions of the compiler. In general, these options should not be set to true unless such compatibility is required. Their default value is false.

Build Configuration options	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

Note: There are several compatibility options that have switches beginning with -Vb

. These options are summarized in the following table:

Switch	Meaning
-Vb	Turn on all -Vb switches.
-Vb+	Turn on all -Vb switches.
-Vb-	Turn off all -Vb switches.
-Vb.	Reset all -Vb switches.
-Vbe	Allow old-style explicit template specialization.
-Vbn	Allow calling a non-const member function for a const object.

-Vbo	Use old overload resolution rules.
-Vbr	Allow non-const reference binding.
-Vbs	Do not treat string literals as const.
-Vbx	Allow explicit template specializations as member functions.

C++ Compatibility options	**Description**
Non-const calls for const object (-Vbn)	Allow calling a non-const member function for a const object. Default = false
Old overload resolution (-Vbo)	Use old overload resolution rules. Default = false
Non-const reference binding (-Vbr)	Allow non-const reference binding. Default = false
Explicit template specialization as member function (-Vbx)	Allow explicit template specializations as member functions. Default = false
Old-style explicit template specialization (-Vbe)	Allow old-style explicit template specialization. Default = false
Old style class arguments (-Va)	Supports old style class arguments. Default = false
Constructor displacements (-Vc)	Supports constructor displacements. Default = false

Old for-statement scoping (-Vd)	Specifies the scope of variables declared in for loop expressions. The output of the following code segment changes, depending on the setting of this option.
	`int main(void)`
	`{`
	`for(int i=0; i<10; i++)`
	`{`
	`cout << "Inside for loop, i = " << i`
	`<< endl;`
	`} //end of for-loop block`
	`cout << "Outside for loop, i = " << i <<`
	`endl; //error without -Vd`
	`} //end of block containing for loop`
	If this option is disabled (the default), the variable*i* goes out of scope when processing reaches the end of the for loop. Because of this, you'll get an Undefined Symbol compilation error if you compile this code with this option disabled.
	If this option is set (-Vd), the variable i goes out of scope when processing reaches the end of the block containing the for loop. In this case, the code output would be:
	`Inside for loop, i = 0`
	`...`
	`Outside for loop, i = 10`
	Default = false

Old Borland class layout (-Vl)	This is a backward compatibility switch that causes the C++ compiler to lay out derived classes the same way it did in older versions of C++Builder. Enable this option if you need to compile source files that you intend to use with older versions of C++Builder (for example, if you need to work with a DLL that you cannot recompile, or if you have older data files that contain hardwired class layouts). Default = false
Push 'this' first (-Vp)	Like Pascal, pushes 'this' first. The compiler typically pushes parameters on the stack from right to left. Default = false
VTable in front (-Vt)	Puts virtual table pointer at front of object layout. Default = false
'Slow' virtual base pointers (-Vv)	Uses 'slow' virtual base pointers. Default = false
Zero-length empty class member functions (-Vx)	Usually the size of a data member in a class definition is at least one byte. When this option is enabled, the compiler allows an empty structure of zero length. Default = false
Zero-length empty base class (-Ve)	Usually the size of a class is at least one byte, even if the class does not define any data members. When you set this option, the compiler ignores this unused byte for the memory layout and the total size of any derived classes; empty base classes do not consume space in derived classes. Default = false

General option	Description
Default	Saves the current settings as the default for each new project.

3.2.11.4.2.3 C++ Compiler C++ Compilation

Project ▶ Options ▶ C++ Compiler ▶ C++ Compilation

Use this dialog box to set C++ Compiler options.

Build Configuration options	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

C++ options	Description
Template generation	**Default** (-Jgd)* The compiler generates public (global) definitions for all template instances. If more than one module generates the same template instance, the linker automatically merges duplicates to produce a single copy of the instance. To generate the instances, the compiler must have available the function body (in the case of a template function) or the bodies of member functions and definitions for static data members (in the case of a template class), typically in a header file. This is a convenient way of generating template instances. This is the default. **External** (-Jgx) The compiler generates external references to all template instances. If you use this option, all template instances that need to be linked must have an explicit instantiation directive in at least one other module.
Virtual tables	**Smart** (-V)* Generates common C++ virtual tables and out-of-line inline functions across the modules in your application. As a result, only one instance of a given virtual table or out-of-line inline function is included in the program. The Smart option generates the smallest and most efficient executables, but produces .OBJ and .ASM files compatible only with CodeGear linkers and assemblers. This is the default. **External** (-V0) Generate external references to virtual tables. If you don't want to use the Smart option, use the External and Public options to produce and reference global virtual tables. When you use this option, one or more of the modules comprising the program must be compiled with the Public option to supply the definitions for the virtual tables. **Public** (-V1) Public produces public definitions for virtual tables. When using the External option (-V0), at least one of the modules in the program must be compiled with the Public option to supply the definitions for the virtual tables. All other modules should be compiled with the External option to refer to that Public copy of the virtual tables.
Member pointers	**Smallest possible** (-Vmd) Member pointers use the smallest possible representation that allows them to point to all members of their particular class. If the class is not fully defined at the point where the member pointer type is declared, the most general representation is chosen by the compiler and a warning is issued. This is the default. **Multiple inheritance** (-Vmm) Member pointers can point to members of multiple inheritance classes (with the exception of virtual base classes). **Single inheritance** (-Vms) Member pointers can point only to members of base classes that use single inheritance. **Default** * No options set for member pointers. This is the default.
Honor member precision (-Vmp)	The compiler uses the declared precision for member pointer types. Use this option when a pointer to a derived class is explicitly cast as a pointer-to-member of a simpler base class (when the pointer is actually pointing to a derived class member). Default = false

Exception handling options	Description
Enable RTTI (-RT)	Causes the compiler to generate code that allows runtime type identification (RTTI). In general, if you set Enable Destructor Cleanup (-xd), you need to set this option as well. Default = true
Enable exceptions (-x)	Sets C++ exception handling. If this option is disabled (-x-) and you attempt to use exception handling routines in your code, the compiler generates error messages during compilation. Disabling this option makes it easier for you to remove exception handling information from programs; this might be useful if you are porting your code to other platforms or compilers. Disabling this option turns off only the compilation of exception handling code; your application can still include exception code if you link object and library files that were built with exceptions enabled (such as the CodeGear runtime libraries). Default = true
Destructor cleanup (-xd)	When this option is set and an exception is thrown, destructors are called for all automatically declared objects between the scope of the `catch` and `throw` statements. In general, when you set this option, you should also set Enable Runtime Type Information (-RT) as well. Destructors are not automatically called for dynamic objects allocated with `new`, and dynamic objects are not automatically freed. Default = true
No DLL/MT destructor cleanup (-xds)	Does not perform DLL or multi-threaded destructor cleanups. Default = false
Fast exception prologs (-xf)	Expands inline code for every exception handling function. This option improves performance at the cost of larger executable file sizes. Default = false
Location information (-xp)	When this option is set, runtime identification of exceptions is available, because the compiler provides the file name and source code line number where the exception occurred. This enables the program to use the **__ThrowFileName** global to obtain the file where the exception occurred and the **__ThrowLineNumber** global to access the line number from where the C++ exception was thrown. Default = false
Slow exception epilogues (-xs)	When this option is set, the exception handling epilogue code is not expanded inline. This option decreases performance slightly. Default = false
Hide exception variables (-xv)	The compiler treats the following exception handling symbols as special: **__exception_info** **__exception_code** **__abnormal_termination** These are all mapped to special compiler/RTL constructs for Structured Exception Handling (SEH) code. If you are not using SEH and you have variables of this name, it means that you could not reference those variables, and your code would not compile. -xv causes the compiler to hide its special symbols in this event, so that you can use variables of this name. Default = false
Global destructor count (-xdg)	Use global destructor count (for compatibility with older versions of the compiler). Default = false

General option	Description
Default	Saves the current settings as the default for each new project.

3.2.11.4.2.4 C++ Compiler Debugging

Project ▶ Options ▶ C++ Compiler ▶ Debugging

Use this dialog box to set C++ compiler debugging and CodeGuard options.

Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

Debugging options	Description
Debug information (-v)	Includes debugging information in your .OBJ files. The compiler passes this option to the linker so it can include the debugging information in the .EXE file. For debugging, this option treats C++ inline functions as normal functions. You need debugging information to use either the integrated debugger or the standalone Turbo Debugger. When this option is off (-v-), you can link and create larger object files. While this option does not affect execution speed, it does affect compilation and link time. When Line Numbers is on, make sure you turn off Pentium scheduling in the Compiler options. When this option is set, the source code does not exactly match the generated machine instructions, which can make stepping through code confusing. Default = false
Debug line number information (-y)	Automatically includes line numbers in the object and object map files. Line numbers are used by both the integrated debugger and Turbo Debugger. Although the Debug Info in OBJs option (-v) automatically generates line number information, you can turn that option off (-v-) and turn on Line Numbers (-y) to reduce the size of the debug information generated. With this setup, you can still step, but you cannot watch or inspect data items. Including line numbers increases the size of the object and map files but does not affect the speed of the executable program. When Line Numbers is on, make sure you turn off Pentium scheduling in the Compiler options. When this option is set, the source code does not exactly match the generated machine instructions, which can make stepping through code confusing. Default = false
Expand inline functions (-vi)*	Expands C++ inline functions inline. To control the expansion of inline functions, the Debug Information In OBJs option (-v) acts slightly different for C++ code: when inline function expansion is disabled, inline functions are generated and called like any other function. Default = true
Generate CodeView4–compatible debug info (-v4)	Generates CodeView4 compatible debug information. Default = false

CodeGuard(tm) options	Description
Enable all CodeGuard options (-vG)	Enables all CodeGuard options, independent of CodeGuard level. Default = false

Monitor inline pointer access (-vGc)	This CodeGuard option generates calls to verify all accesses in your code. This option identifies almost all pointer errors. Program execution is typically five to ten times slower. Selecting any of these CodeGuard options can have a noticeable effect on runtime performance. Default = false
Monitor global and stack data accesses (-vGd)	Creates data and stack layout descriptors for fast lookup by CodeGuard. These descriptors allow CodeGuard to report overruns and invalid pointers to locals, globals, and statics. You should always use this option. Default = false
Monitor 'this' pointer on member function entry (-vGt)	Creates special epilogs for member functions, constructors, and destructors. CodeGuard verifies the **this** pointer on entry to every method in C++ code. This option is useful because it reports calls to methods of deleted or invalid objects even if the methods themselves do not access **this**. Default = false

General option	Description
Default	Saves the current settings as the default for each new project.

Note: If you want to turn both debugging and inline expansion on, use the -v and -vi options.

3.2.11.4.2.5 C++ Compiler General Compatibility

Project ▶ Options ▶ C++ Compiler ▶ General Compatibility

Use this dialog box to set C++ Compiler General Compatibility options.

Some of these options provide backward compatibility with previous versions of the compiler, and their default value is false. In general, such options should not be set to true unless such compatibility is required.

Build Configuration options	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

Note: There are several compatibility options that have switches beginning with -Vb

, summarized in this table:

Switch	Meaning
-Vb	Turn on all -Vb switches.
-Vb+	Turn on all -Vb switches.
-Vb-	Turn off all -Vb switches.
-Vb.	Reset all -Vb switches.
-Vbe	Allow old-style explicit template specialization.
-Vbn	Allow calling a non-const member function for a const object.
-Vbo	Use old overload resolution rules.
-Vbr	Allow non-const reference binding.

-Vbs	Do not treat string literals as const.
-Vbx	Allow explicit template specializations as member functions.

General Compatibility options	**Description**
Non-const string literals (-Vbs)	Do not treat string literals as const. Default = false
Global functions in segments (-VA)*	Generates all global functions in their own virtual or weak segment. Default = true
Don't mangle calling convention (-VC)	When this option is set, the compiler disables the distinction of function names where the only possible difference is incompatible code generation options. For example, with this option set, the linker does not detect if a call is made to a **__fastcall** member function with the **cdecl** calling convention. This option is provided for backward compatibility only; it lets you link old library files that you cannot recompile. Default = false
Microsoft header search algorithm (-VI)*	Uses Microsoft search algorithms to locate the header files. Default = true
VC++ compatibility (-VM)	To provide compatibility with Microsoft Visual C++ , substitutes __msfastcall for __fastcall calling convention. This switch should **not** be used when working with a VCL application. It causes numerous linker errors. Default = false
Disable lexical digraph scanner (-Vg)	Disables the lexical digraph scanner. Digraphs are two character sequences that stand in for a single character that may be hard to produce on certain keyboards. If this option is true, then such diagraphs are not recognized. Default = false
Enable new operator names (-Vn)	Enables new operator names such as 'and', 'or', 'and_eq', 'bitand', etc. Default = false
Enable all compatibility options (-Vo)	Sets most of the compatibility flags used with old code, enabling -Vv , -Va, -Vp, -Vt, -Vc, -Vd, and -Vx. Default = false
Reverse Multi-character constants (-Vr)	The compiler reverses the order of Multi-character constants. Default = false
Old style virdef generation (-Vs)	Uses old-style virdef generation. Default = false
Native code for MBCS (-Vw)	Emits native code instead of Unicode for multi-byte character. Default = false
Old 8.3 include search (-Vi)	Use old 8.3 search algorithm to locate header files. Default = false

General option	**Description**
Default	Saves the current settings as the default for each new project.

3.2.11.4.2.6 C++ Compiler General Compilation

Project ▶ Options ▶ C++ Compiler ▶ General Compilation

Use this dialog box to set C++ Compiler General Compilation options.

Build Configuration options	Description
Build Configuration	Displays the active build configuration. You can use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

Compilation options	Description
Instruction Set	**80386** (-3)* Generates 80386 protected-mode compatible instructions. This is the default. **80486** (-4) Generates i486 protected-mode compatible instructions. **Pentium** (-5) Generates Pentium instructions. While this option increases the speed at which the application runs on Pentium machines, expect the program to be a bit larger than when compiled with the 80386 or i486 options. In addition, Pentium-compiled code sustains a performance hit on non-Pentium systems. **Pentium Pro** (-6) Generates Pentium Pro instructions.
Data Alignment	**Byte** (-a1) Does not force alignment of variables or data fields to any specific memory boundaries. The compiler aligns data at even or odd addresses, depending on the next available address. While byte alignment produces more compact programs, the programs tend to run slower. The other data alignment options increase the speed at which 80x86 processors fetch and store data. **Word** (-a2) 2 byte data alignment. Aligns non-character data at even addresses. Automatic and global variables are aligned properly. **char** and **unsigned char** variables and fields can be placed at any address; all others are placed at even-numbered addresses. **Double word** (-a4)* 4 byte data alignment. Aligns non-character data at 32–bit word (4–byte) boundaries. Data type sizes of less than four bytes are aligned on their type size. This is the default. **Quad word** (-a8) 8 byte data alignment. Aligns non-character data at 64–bit word (8–byte) boundaries. Data with type sizes of less than eight bytes are aligned on their type size. **Paragraph** (-a16) 16 byte data alignment. Aligns non-character data at 128–bit (16–byte) boundaries. Data with type sizes of less than 16 bytes are aligned on their type size.

Register variables	**None** (-r-)* Disables the use of register variables. Tells the compiler not to use register variables, even if you have used the **register** keyword. This is the default. **Explicit** (-rd) Use register variables only if you use the **register** keyword and a register is available. Use this option or the **Always** option (-r) to optimize the use of registers. You can use -rd in **#pragma** options. **Always** (-r) Automatically assign register variables if possible, even when you do not specify a register variable by using the **register** keyword. Generally, you can use **Always**, unless you are interfacing with preexisting assembly code that does not support register variables.
Calling convention	**Pascal (-p)** Tells the compiler to generate a Pascal calling sequence for function calls (do not generate underbars, all uppercase, calling function cleans stack, pushes parameters left to right). This is the same as declaring all subroutines and functions with the **__pascal** keyword. The resulting function calls are usually smaller and faster than those made with the C (-pc) calling convention. Functions must pass the correct number and type of arguments. You can use the **__cdecl**, **__fastcall**, or **__stdcall** keywords to specifically declare a function or subroutine using another calling convention. **C (-pc)*** Tells the compiler to generate a C calling sequence for function calls (generate underbars, case sensitive, push parameters right to left). This is the same as declaring all subroutines and functions with the **__cdecl** keyword. Functions declared using the C calling convention can take a variable parameter list (the number of parameters does not need to be fixed). You can use the **__pascal**, **__fastcall**, or **__stdcall** keywords to specifically declare a function or subroutine using another calling convention. This is the default. **_msfastcall (-pm)** Tells the compiler to substitute the **__msfastcall** calling convention for any function without an explicitly declared calling convention. **Fastcall (register) (-pr)** Forces the compiler to generate all subroutines and all functions using the Register parameter-passing convention, which is equivalent to declaring all subroutine and functions with the **__fastcall** keyword. With this option enabled, functions or routines expect parameters to be passed in registers. You can use the **__pascal**, **__cdecl**, or **__stdcall** keywords to specifically declare a function or subroutine using another calling convention. **stdcall (-ps)** Tells the compiler to generate a stdcall calling sequence for function calls (does not generate underscores, preserve case, called function pops the stack, and pushes parameters right to left). This is the same as declaring all subroutines and functions with the **__stdcall** keyword. Functions must pass the correct number and type of arguments. You can use the **__cdecl**, **__pascal**, **__fastcall** keywords to specifically declare a function or subroutine using another calling convention.

Compliance	**ANSI** (-A) Use ANSI keywords and extensions. Compiles C and C++ ANSI-compatible code, allowing for maximum portability. Non-ANSI keywords are ignored as keywords. **K & R** (-AK) Use Kernighan and Ritchie (KR) keywords and extensions. Tells the compiler to recognize only the KR extension keywords and treat any of CodeGear's C++ extension keywords as normal identifiers. **Borland** (also -A-) (-AT)* Use Borland/CodeGear C++ keywords and extensions. Tells the compiler to recognize CodeGear's extensions to the C language keywords, including **near**, **far**, **huge**, **asm**, **cdecl**, **pascal**, **interrupt**, **_export**, **_ds**, **_cs**, **_ss**, **_es**, and the register pseudovariables (**_AX**, **_BX**, and so on). This is the default. **Unix System V** (-AU) Use UNIX System V keywords and extensions. Tells the compiler to recognize only UNIX V keywords and treat any of CodeGear's C++ extension keywords as normal identifiers. **Hint:**If you get declaration syntax errors from your source code, check that this option is set to Borland Extensions.
Extended error info (-Q)	Compiler generates more extended information on errors. (Default = false)
Standard stack frames (-k)	Generates a standard stack frame (standard function entry and exit code). This is helpful when debugging, since it simplifies the process of stepping through the stack of called subroutines. When this option is off, any function that does not use local variables and has no parameters is compiled with abbreviated entry and return code. This makes the code smaller and faster. The Standard Stack Frame option should always be on when you compile a source file for debugging. (Default = false)
Integer-sized enums (-b)	Allocates a whole word (a four-byte int for 32–bit programs) for enumeration types (variables of type enum). When this option is off (-b-), the compiler allocates the smallest integer that can hold the enumeration values: the compiler allocates an unsigned or signed char if the values of the enumeration are within the range of 0 to 255 (minimum) or -128 to 127 (maximum), or an unsigned or signed short if the values of the enumeration are within the following ranges: 0..65535 or -32768..32767. The compiler allocates a four-byte int (32-bit) to represent the enumeration values if any value is out of these ranges. (Default = true)
Treat warnings as errors (-w!)	Causes the compiler to treat warnings as errors. (Default = false)
Force C++ compile (-P)	Causes the compiler to compile all source files as C++ files, regardless of their extension. (Default = false)
Batch compile	Enable batch file compile. (Default = false)

General option	Description
Default	Saves the current settings as the default for each new project.

3.2.11.4.2.7 C++ Compiler

Project ▶ Options ▶ C++ Compiler

This is the top-level node of the C++ Compiler command line options.

Note: Options marked with an asterisk (*) on the options pages are the default values.

Build Configuration option	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

General option	Description
Default	Saves the current settings as the default for each new project.

See Also

General Compilation (see page 858)

Advanced Compilation (see page 849)

C++ Compilation (see page 853)

General Compatibility (see page 857)

C++ Compatibility (see page 851)

Debugging (see page 856)

Output (see page 863)

Optimizations (see page 862)

Paths and Defines (see page 864)

Precompiled headers (see page 865)

Warnings (see page 866)

3.2.11.4.2.8 C++ Compiler Optimizations

Project ▸ Options ▸ C++ Compiler ▸ Optimizations

Use this dialog box to set C++ Compiler Optimization options.

Build Configuration option	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

Optimizations options	Description
None (-Od)*	Disables all optimization settings, including ones which you may have specifically set and those that would normally be performed as part of the speed/size trade-off. Because this disables code compaction (tail merging) and cross-jump optimizations, using this option can keep the debugger from jumping around or returning from a function without warning, which makes stepping through code easier to follow. This is the default.
Size (-O1)	Sets an aggregate of optimization options that tells the compiler to optimize your code for size. For example, the compiler scans the generated code for duplicate sequences. When such sequences warrant, the optimizer replaces one sequence of code with a jump to the other and eliminates the first piece of code. This occurs most often with `switch` statements. The compiler optimizes for size by choosing the smallest code sequence possible.
Speed (-O2)	This switch sets an aggregate of optimization options that tells the compiler to optimize your code for speed.
Selected	Chooses specific optimizations to enable. Click **Select All** to enable all optimizations in the list. Click **Clear All** to clear all selected optimizations in the list Click **Defaults** to enable the default optimizations in the list

General option	Description
Default	Saves the current settings as the default for each new project.

3.2.11.4.2.9 C++ Compiler Output

Project ▶ Options ▶ C++ Compiler ▶ Output

Use this dialog box to set C++ Compiler Output options.

Build Configuration options	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

.obj Content options	Description
Disable compiler autodependency output (-X)	Disable compiler autodependency output. Default = false
Exclude system headers from dependency info (-mm)	Ignores system header files while generating dependency information. Default = false

Generate underscores on symbol names (-u)*	The compiler automatically adds an underscore character (_) in front of every global identifier (functions and global variables) before saving them in the object module. Pascal identifiers (those modified by the **__pascal** keyword) are converted to uppercase and are not prefixed with an underscore. Underscores for C and C++ are optional, but you should turn this option on to avoid errors if you are linking with the CodeGear C++ libraries. Default = true
Don't prefix underbars to exported symbols (-vu)	Do not prefix underscore characters to exported symbol names. Default = false
Include browser information .obj files (-R)	Includes browser information in generated .OBJ files. Default = false

General option	Description
Default	Saves the current settings as the default for each new project.

3.2.11.4.2.10 C++ Compiler Paths And Defines

Project ▶ Options ▶ C++ Compiler ▶ Paths and Defines

Use this dialog box to set C++ Compiler Paths and Defines options.

Build Configuration options	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

Paths and Defines options	Description
Include path (-I)	Specifies the drive and/or directories that contain program include files. Standard include files are those you specify in angle brackets (<>) in an #include statement (for example, #include <myfile>). Click [...] to display a dialog allowing you to edit a list of search paths. Check the **Merge** box to act as if the immediate ancestor's paths are merged into this list, though this list is not actually changed.
Defines (-D)	Defines the specified identifier name to the null string. -D*name=string* defines name to string. In this assignment, string cannot contain spaces or tabs. You can also define multiple **#define** options on the command line using either of the following methods: Include multiple definitions after a single -D option by separating each define with a semicolon (;) and assigning values with an equal sign (=). For example: `BCC32.EXE -Dxxx;yyy=1;zzz=NO MYFILE.C` Include multiple -D options, separating each with a space. For example: `BCC32.EXE -Dxxx -Dyyy=1 -Dzzz=NO MYFILE.C` Click [...] to display a dialog allowing you to edit a list of defines. Check the **Merge** box to act as if the immediate ancestor's defines are merged into this list, though this list is not actually changed.

.obj output directory (-n)	Sets output directory to specified path. Click [...] to browse for a folder.
Windows version target	Conditional defines targeting the highest version of Windows API header files to use. Choose an OS version from the drop down list. See http://msdn2.microsoft.com/en-us/library/aa383745.aspx for more information. The default is “Not specified”.

General option	Description
Default	Saves the current settings as the default for each new project.

3.2.11.4.2.11 C++ Compiler Precompiled Headers

Project ▶ Options ▶ C++ Compiler ▶ Precompiled headers

Use this dialog box to set C++ Compiler Precompiled headers options.

Build Configuration options	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

Usage options	Description
Usage	**Do not use** Do not use precompiled headers. **Generate and use** (-H)* The IDE generates and uses precompiled headers. The default file name is <*projectname*>.CSM for IDE projects, and is BC32DEF.CSM for command-line compiles. This is the default. **Use but don't generate** (-Hu) Compilers use preexisting precompiled header files; new precompiled header files are not generated.
PCH filename (-H=)	Specify the name of your precompiled header file. The compiler sets the name of the precompiled header to the specified filename. Click [...] to display a file dialog to select a file. When this option is set, the compilers generate and use the precompiled header file that you specify.
Cache precompiled headers (-Hc)	The compiler caches the precompiled headers it generates. This is useful when you are precompiling more than one header file. Default = false
Enable smart cached precompiled headers (-Hs)*	The compiler smart-caches the precompiled headers it generates (smart-caching uses less memory than the regular caching option -Hc). Caching header files in memory is useful when you are precompiling more than one header file. Default = true

Warning: If you import a project from BDS2006, it does not import the project's PCH file location due to Windows Vista compatibility, since Vista restricts where users may place files.

Generation options	Description
Replace header names: (-Hr)	Replaces header name from *name1* to *name2*. Click [...] to display a dialog that allows you to manage a list of header files.
Stop precompiling after: (-Hh=)	Terminates compiling the precompiled header after processing the specified file. You can use this option to reduce the disk space required for precompiled headers. Click [...] to display a file selection dialog. When you use this option, the file you specify must be included from a source file for the compiler to generate a .CSM file. You can also use **#pragma hdrstop** within your .CPP files to specify when to stop the generation of precompiled headers. You cannot specify a header file that is included from another header file. For example, you cannot list a header included by windows.h, because this would cause the precompiled header file to be closed before the compilation of windows.h was completed.
Include header content (-Hi)	Includes the contents of the specified header file(s). Click [...] to display a dialog that allows you to manage a list of header files. Check the **Merge** box to act as if the immediate ancestor's files are merged into this list, though this list is not actually changed.
Enable PCH with external type files (-He)*	The compiler generates a file or files that contain debug type information for all the symbols contained in the precompiled headers. The files end with the *.#xx* extension, where *xx* is 00 for the first file generated and is incremented for each additional type-information file required. Using this option dramatically decreases the size of your .OBJ files, since debug type information is centralized and is not duplicated in each .OBJ file. Default = true

General option	Description
Default	Saves the current settings as the default for each new project.

3.2.11.4.2.12 C++ Compiler Warnings

Project ▶ Options ▶ C++ Compiler ▶ Warnings

Use this dialog box to set C++ Compiler Warning options.

Build Configuration options	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

Warnings options	Description
Enable all (-w)	Display all warning and error messages.
Disable all (-w-)	Disable all warning and error messages

Selected *	Choose specific warnings to enable. This is the default. Click **Select All** to display all warnings in the list. Click **Clear All** to clear all selected warnings in the list Click **Defaults** to display the default warnings in the list

General option	Description
Default	Saves the current settings as the default for each new project.

3.2.11.4.3 Folder or Directory View

Use this dialog box to add a folder node to the active project. You can use a folder node or a directory view to browse frequently used files that are not part of your project.

Item	Description
Location	Specifies the path of the new folder node. The location can be an absolute path or a relative path. You can click **[...]** to browse to a folder.
File Types	Specifies the filter that you can use to hide unwanted files. By default, the **File Types** filter is `*.*`, which displays all files.
Show Subdirectories	Displays subdirectories in the folder node. If this option is disabled, only files are shown.

3.2.11.4.4 Resource Compiler

Topics

Name	Description
Resource Compiler (see page 867)	**Project ▶ Options ▶ Resource Compiler** This is the top-level node of the Resource Compiler command line options. **Note:** Options marked with an asterisk (*) on the options pages are the default values.
Resource Compiler Options (see page 868)	**Project ▶ Options ▶ Resource Compiler ▶ Options** Use this dialog box to set Resource Compiler options.
Resource Compiler Paths And Defines (see page 869)	**Project ▶ Options ▶ Resource Compiler ▶ Paths and Defines** Use this dialog box to set Resource Compiler Paths and Defines options.

3.2.11.4.4.1 Resource Compiler

Project ▶ Options ▶ Resource Compiler

This is the top-level node of the Resource Compiler command line options.

Note: Options marked with an asterisk (*) on the options pages are the default values.

Build Configuration option	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.

Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

General option	Description
Default	Saves the current settings as the default for each new project.

See Also

Paths and Defines (↗ see page 869)

Options (↗ see page 868)

3.2.11.4.4.2 Resource Compiler Options

Project ▶ Options ▶ Resource Compiler ▶ Options

Use this dialog box to set Resource Compiler options.

Build Configuration option	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

Options	Description
Resource type	**Windows 3.1** (-31 Builds Windows 3.1-compatible .RES files. **16 bit** (-16) Builds a 16-bit resource. **32 bit** (-32)* Builds a 32-bit resource. This is the default.
Default language (-l)	Specifies the default language. For example, `–l409` represents English. See http://msdn2.microsoft.com/en-us/library/ms776324.aspx for more information about specifying language identifiers.
Code page (-c)	Uses the specified code page for resource translation. If -c is not used, the default ANSI code page is used.
Additional options	Enter additional options for the resource compiler.
Ignore INCLUDE (-x)	Ignore INCLUDE environment variable. Default = false
Verbose messages (-v)	The linker emits detailed information messages. Default = false
Multi-byte character support (-m)	Multi-byte character support. Default = false

General option	Description
Default	Saves the current settings as the default for each new project.

3.2.11.4.4.3 Resource Compiler Paths And Defines

Project ▶ Options ▶ Resource Compiler ▶ Paths and Defines

Use this dialog box to set Resource Compiler Paths and Defines options.

Build Configuration option	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

Paths and Defines options	Description
Include path: (-I)	Specifies the drive and/or directories that contain program include files. Standard include files are those you specify in angle brackets (<>) in an #include statement (for example, #include <myfile>). Click [...] to display a **Include file search path** dialog to manage a list of paths. Check the **Merge** box to act as if the immediate ancestor's paths are merged into this list, though this list is not actually changed.
Defines: (-d)	Defines list of preprocessor symbols. Click [...] to display a **Defines a preprocessor symbol** dialog to manage a list of preprocessor symbols. Check the **Merge** box to act as if the immediate ancestor's defines are merged into this list, though this list is not actually changed.
.obj output directory	Sets the .obj output directory to the specified directory. Click [...] to display a directory selection dialog.

General option	Description
Default	Saves the current settings as the default for each new project.

3.2.11.4.5 DCC32

Topics

Name	Description
Delphi Compiler Compiling (↗ see page 870)	**Project ▶ Options ▶ Delphi Compiler ▶ Compiling** Use this dialog box to set Delphi Compiler Compiling options.
Delphi Compiler (↗ see page 874)	**Project ▶ Options ▶ Delphi Compiler** This is the top-level node of the C++ Compiler command line options. **Note:** Options marked with an asterisk (*) on the options pages are the default values.
Delphi Compiler Other Options (↗ see page 875)	**Project ▶ Options ▶ Delphi Compiler ▶ Other Options** Use this dialog box to set Delphi Compiler Other options.

Delphi Compiler Paths and Defines (↗ see page 876)	**Project ▶ Options ▶ Delphi Compiler ▶ Paths and Defines** Use this dialog box to set CodeGear Pascal Compiler Path and Define options.
Delphi Compiler Warnings (↗ see page 877)	**Project ▶ Options ▶ Delphi Compiler ▶ Warnings** Use this dialog box to set Delphi Compiler Warning options.

3.2.11.4.5.1 Delphi Compiler Compiling

Project ▶ Options ▶ Delphi Compiler ▶ Compiling

Use this dialog box to set Delphi Compiler Compiling options.

Build Configuration option	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

Syntax options	Description
Strict var-strings (-$V+)*	This option (equivalent to $V directive) is meaningful only for Delphi code that uses short strings and is provided for backwards compatibility with early versions of Delphi and Borland Pascal. The option controls type checking on short strings passed as variable parameters. When enabled (equivalent to {$V+}), strict type checking is performed, requiring the formal and actual parameters to be of identical string types. When disabled (equivalent to {$V-}) (relaxed), any short string type variable is allowed as an actual parameter, even if the declared maximum length is not the same as that of the formal parameter. Default = true
Full boolean evaluation (-$B+)	Switches between the two different models of Delphi code generation for the AND and OR Boolean operators. When enabled (equivalent to {$B+}), the compiler generates code for complete Boolean expression evaluation. This means that every operand of a Boolean expression built from the AND and OR operators is guaranteed to be evaluated, even when the result of the entire expression is already known. When disabled (equivalent to {$B-}), the compiler generates code for short-circuit Boolean expression evaluation, which means that evaluation stops as soon as the result of the entire expression becomes evident in left to right order of evaluation. Default = false
Extended syntax (-$X+)*	Provided for backward compatibility. You should not use this option (equivalent to {$X-} mode) when writing Delphi applications. This option enables or disables Delphi's extended syntax: **Function statements.** In the {$X+} mode, function calls can be used as procedure calls; that is, the result of a function call can be discarded, rather than passed to another function or used in an operation or assignment. Generally, the computations performed by a function are represented through its result, so discarding the result makes little sense. Sometimes, however, a function is called because it performs a task such as setting the value of a global variable, without producing a useful result. **The Result variable.** When enabled (equivalent to {$X+}, the predefined variable Result can be used within a function body to hold the function's return value. **Null-terminated strings.** When enabled, Delphi strings can be assigned to zero-based character arrays (array[0..X] of Char), which are compatible with PChar types. Default = true

Typed '@' operator (-$T+)	Controls the types of pointer values generated by the @ operator and the compatibility of pointer types. When disabled (equivalent to {$T-}), the result of the @ operator is always an untyped pointer (Pointer) that is compatible with all other pointer types. When @ is applied to a variable reference in the enabled (equivalent to {$T+}), the result is a typed pointer that is compatible only with Pointer and with other pointers to the type of the variable. When disabled, distinct pointer types other than Pointer are incompatible (even if they are pointers to the same type). When enabled, pointers to the same type are compatible. Default = false
Open string parameters (-$P+)*	Meaningful only for code compiled supporting huge strings, and is provided for backwards compatibility with early versions of Delphi and Borland Pascal. This option, (equivalent to $P) controls the meaning of variable parameters declared using the string keyword in the huge strings disabled (equivalent to {$H-}) state. When disabled (equivalent to {$P-}), variable parameters declared using the string keyword are normal variable parameters, but when enabled (equivalent to {$P+}), they are open string parameters. Regardless of the setting of this option, the openstring identifier can always be used to declare open string parameters. Default = true
Long strings by default (-$H+)*	Delphi for Win32 only. This option (equivalent to the $H directive) controls the meaning of the reserved word string when used alone in a type declaration. The generic type string can represent either a long, dynamically-allocated string (the fundamental type AnsiString) or a short, statically allocated string (the fundamental type ShortString). By default, Delphi defines the generic string type to be the long AnsiString. All components in the component libraries are compiled in this state. If you write components, they should also use long strings, as should any code that receives data from component library string-type properties. The disabled (equivalent to {$H-}) state is mostly useful for using code from versions of Delphi that used short strings by default. You can locally override the meaning of string-type definitions to ensure generation of short strings. You can also change declarations of short string types to string[255] or ShortString, which are unambiguous and independent of the enabled option. Default = true
Writeable structured constants (-$J+)	Controls whether typed constants can be modified or not. When enabled (equivalent to {$J+}), typed constants can be modified, and are in essence initialized variables. When disabled (equivalent to {$J-}), typed constants are truly constant, and any attempt to modify a typed constant causes the compiler to report an error. Writeable consts refers to the use of a typed const as a variable modifiable at runtime. Old source code that uses writeable typed constants must be compiled with this option enabled, but for new applications it is recommended that you use initialized variables and compile your code with the option disabled. Default = false

Debugging options	Description
Debug information (-$D+)	Enables or disables the generation of debug information. This information consists of a line-number table for each procedure, which maps object-code addresses into source text line numbers. For units, the debug information is recorded in the unit file along with the unit's object code. Debug information increases the size of unit file and takes up additional memory when compiling programs that use the unit, but it does not affect the size or speed of the executable program. When a program or unit is compiled with this option enabled (equivalent to {$D+}), the integrated debugger lets you single-step and set breakpoints in that module. The **Full debug information** and **Map file** options (on the **Linker** pages of the **Project Options** dialog) produce complete line information for a given module only if you've compiled that module with this option set on. This option is usually used in conjunction with the **Local symbols** option (the $L switch), which enables and disables the generation of local symbol information for debugging. Default = false

Local debug symbols (-$L+)*	Enables or disables the generation of local symbol information. Local symbol information consists of the names and types of all local variables and constants in a module, that is, the symbols in the module's implementation part and the symbols within the module's procedures and functions. For units, the local symbol information is recorded in the unit file along with the unit's object code. Local symbol information increases the size of unit files and takes up additional memory when compiling programs that use the unit, but it does not affect the size or speed of the executable program. When a program or unit is compiled with this option enabled (equivalent to {$L+}), the integrated debugger lets you examine and modify the module's local variables. Furthermore, calls to the module's procedures and functions can be examined by way of the View|Call Stack. The Include TD32 debug info and Map file options (on the **Linker** page of the **Project Options** dialog) produce local symbol information for a given module only if that module was compiled with this option set on. This option is usually used in conjunction with the **Debug information** option, which enables and disables the generation of line-number tables for debugging. This option is ignored if the compiler has the **Debug information** option disabled. Default = true
Assertions (-$C+)*	Enables or disables the generation of code for assertions in a Delphi source file. The option is enabled (equivalent to {$C+}) by default. Since assertions are not usually used at runtime in shipping versions of a product, compiler directives that disable the generation of code for assertions are provided. Uncheck this option to disable assertions. Default = true
Reference info	**None** Use no reference info. **Definitions only** -$DEFINITIONINFO ON* This is the default. **Reference info** -$REFERENCEINFO ON

Runtime error checks options	Description
Range checking (-$R+)	Enables or disables the generation of range-checking code. When enabled, (equivalent to {$R+}), all array and string-indexing expressions are verified as being within the defined bounds, and all assignments to scalar and subrange variables are checked to be within range. If a range check fails, an ERangeError exception is raised (or the program is terminated if exception handling is not enabled). Enabling range checking slows down your program and makes it somewhat larger. Default = false
I/O checking (-$I+)	Enables or disables the automatic code generation that checks the result of a call to an I/O procedure. If an I/O procedure returns a nonzero I/O result when this switch is on, an EInOutError exception is raised (or the program is terminated if exception handling is not enabled). When this switch is off, you must check for I/O errors by calling IOResult. Default = true
Overflow checking (-$Q+)	Controls the generation of overflow checking code. When enabled (equivalent to {$Q+}), certain integer arithmetic operations (+, -, *, Abs, Sqr, Succ, Pred, Inc, and Dec) are checked for overflow. The code for each of these integer arithmetic operations is followed by additional code that verifies that the result is within the supported range. If an overflow check fails, an EIntOverflow exception is raised (or the program is terminated if exception handling is not enabled). This switch is usually used in conjunction with the range checking option ($R switch), which enables and disables the generation of range-checking code. Enabling overflow checking slows down your program and makes it somewhat larger. Default = false

Code generation options	**Description**
Optimization (-$O+)*	Controls code optimization. When enabled (equivalent to {$O+}), the compiler performs a number of code optimizations, such as placing variables in CPU registers, eliminating common subexpressions, and generating induction variables. When disabled, (equivalent to {$O-}), all such optimizations are disabled. Other than for certain debugging situations, you should never need to turn optimizations off. All optimizations performed by the Delphi compiler are guaranteed not to alter the meaning of a program. In other words, the compiler performs no "unsafe" optimizations that require special awareness by the programmer. This option can only turn optimization on or off for an entire procedure or function. You can’t turn optimization on or off for a single line or group of lines within a routine. Default = true
Generate stack frames (-$W+)	Delphi for Win32 only. Controls the generation of stack frames for procedures and functions. When enabled, (equivalent to {$W+}), stack frames are always generated for procedures and functions, even when they're not needed. When disabled, (equivalent to {$W-}), stack frames are only generated when they're required, as determined by the routine's use of local variables. Some debugging tools require stack frames to be generated for all procedures and functions, but other than that you should never need to enable this option. Default = false
Pentium(tm)-safe divide (-$U+)	Delphi for Win32 only. Controls generation of floating-point code that guards against the flawed FDIV instruction exhibited by certain early Pentium processors. Windows 95, Windows NT 3.51, and later Windows OS versions contain code that corrects the Pentium FDIV bug system-wide. When enabled (equivalent to {$U+}), all floating-point divisions are performed using a runtime library routine. The first time the floating-point division routine is invoked, it checks whether the processor's FDIV instruction works correctly, and updates the TestFDIV variable (declared in the System unit) accordingly. For subsequent floating-point divide operations, the value stored in TestFDIV is used to determine what action to take. **-1** means that FDIV instruction has been tested and found to be flawed. **0** means that FDIV instruction has not yet been tested. **1** means that FDIV instruction has been tested and found to be correct. For processors that do not exhibit the FDIV flaw, enabling this option results in only a slight performance degradation. For a flawed Pentium processor, floating-point divide operations may take up to three times longer in the enabled state, but they always produce correct results. In the disabled (equivalent to {$U-}) state, floating-point divide operations are performed using in-line FDIV instructions. This results in optimum speed and code size, but may produce incorrect results on flawed Pentium processors. You should use the disabled state only in cases where you are certain that the code is not running on a flawed Pentium processor. Default = false

Record alignment	**Off** (-$A-) Fields are aligned using the compiler defaults. **Byte** (-$A1) Fields are never aligned. All record and class structures are packed. **Word** (-$A2) Fields in record types that are declared without the packed modifier and fields in class structures are aligned on word boundaries. **Double word** (-$A4) Fields in record types that are declared without the packed modifier and fields in class structures are aligned on double-word boundaries. **Quad word** (-$A8)* Fields in record types that are declared without the packed modifier and fields in class structures are aligned on quad word boundaries. Regardless of the state of the $A directive, variables and typed constants are always aligned for optimal access. By setting the option to-$A8, execution is faster. This is the default.
Minimum enum size	**Byte** (-$Z1) The minimum enum size is 1 byte. **Word** (-$Z2) The minimum enum size is 2 bytes. **Double word** (-$Z4)* The minimum enum size is 4 bytes. This is the default. **Quad word** (-$Z8) The minimum enum size is 8 bytes.
Codepage (—codepage)	Enter the codepage for your application's language. Codepage is a decimal number representing a specific character encoding table, and there are standard values for various languages. Default = 0, no code page

General item	**Description**
Default	Saves the current settings as the default for each new project.

General option	**Description**
Default	Saves the current settings as the default for each new project.

3.2.11.4.5.2 Delphi Compiler

Project ▶ Options ▶ Delphi Compiler

This is the top-level node of the C++ Compiler command line options.

Note: Options marked with an asterisk (*) on the options pages are the default values.

Options marked with an asterisk (*) on the options pages are the default values.

Build Configuration option	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

General option	Description
Default	Saves the current settings as the default for each new project.

See Also

Paths and Defines (↗ see page 876)

Compiling (↗ see page 870)

Other options (↗ see page 875)

Warnings (↗ see page 877)

3.2.11.4.5.3 Delphi Compiler Other Options

Project ▶ Options ▶ Delphi Compiler ▶ Other Options

Use this dialog box to set Delphi Compiler Other options.

Build Configuration option	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

Other Options	Description
Unit aliases: (-A)	Sets unit aliases to the specified aliases.
Use these packages when compiling: (-LU)	Dynamically link with the specified packages.
Additional options:	Additional switches to pass to the compiler.

C/C++ Output options	Description
Object files and headers	**No C/C++ output*** No intermediate output specified. This is the default. **C .objs** (-J) Generates C .obj files. **C++ .objs** (-JP) Generates C++ .obj files. **C++ .objs, headers** (-JPH) Generates C++ .obj and .hpp files. **C++ .objs, headers, namespaces** (-JPHN) Generates C++ .obj and .hpp files including namespaces. **C++ .objs, headers, namespaces, export** (-JPHNE) Generates C++ .obj and .hpp files with namespaces and export symbols. **C++ .objs, namespaces** (-JPN) Generates C++ .obj files with namespaces. **C++ .objs, namespaces, export** (-JPNE) Generates C++ .obj files with namespaces and export symbols. **C++ .objs, headers, exports** (-JPHE) Generates C++ .obj and .hpp files and export symbols. **C++ .objs, exports** (-JPE) Generates C++ .obj and export symbols. **Generate all C++ Builder files (including package libs)** Generates C++ .obj and .hpp files with namespaces and export symbols and package.
Header file output (-NH)	Sets the .hpp output directory to the specified directory. Click [...] to display a **Browse for Folder** dialog.

General item	Description
Default	Saves the current settings as the default for each new project.

3.2.11.4.5.4 Delphi Compiler Paths and Defines

Project ▶ Options ▶ Delphi Compiler ▶ Paths and Defines

Use this dialog box to set CodeGear Pascal Compiler Path and Define options.

Build Configuration option	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

Paths and Defines options	Description
Include path: (-I)	Includes the specified search paths. Click [...] to display the **Include path** dialog to manage a list of defines. Check the **Merge** box to act as if the immediate ancestor's paths are merged into this list, though this list is not actually changed.
Defines: (-D)	Defines a conditional symbol. The directive **-Dsymbol** defines **symbol**. The defined symbol can be used by the **{$IFDEF symbol}** or **{$IFNDEF symbol}** directives or **DEFINED symbol** in the conditional expression part of a **{$IFC cond-expr}** directive. You can also define a symbol by using the **{$DEFINE symbol}** directive in the source file. Include multiple defines after a single **-D** option by separating each define with a semicolon (;). For example: `DCC32.EXE -Dxxx;yyy;zzz MYFILE.PAS` You can also include multiple **-D** options, separating each with a space. For example: `DCC32.EXE -Dxxx -Dyyy -Dzzz MYFILE.PAS` Click [...] to display a **Defines symbol** dialog to manage a list of defines. Check the **Merge** box to act as if the immediate ancestor's defines are merged into this list, though this list is not actually changed.
.obj output directory (-NO)	Sets the .obj output directory to the specified directory. Click [...] to display a directory selection dialog.

General option	Description
Default	Saves the current settings as the default for each new project.

3.2.11.4.5.5 Delphi Compiler Warnings

Project ▶ Options ▶ Delphi Compiler ▶ Warnings

Use this dialog box to set Delphi Compiler Warning options.

Build Configuration option	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

Warnings options	Description
Hints (-H)*	Outputs hint messages. Default = true
Warnings (-W)*	Outputs warning messages. Default = true
Selected warnings	Chooses specific hints and warnings to enable. Click **Select All** to display all hints and warnings in the list. Click **Clear All** to clear all selected hints and warnings in the list Click **Defaults** to display the default hints and warnings in the list

General option	Description
Default	Saves the current settings as the default for each new project.

3.2.11.4.6 Find Option

Project ▶ Options

Use this dialog box to find a specific option for the selected build tool.

Item	Description
Search for	Specifies the search criteria. You can search for an option by typing its description, such as `data alignment`, or its command-line switch, such as `-a`.
Options list	Lists all command-line options that match the search text. When you select an option and click **OK**, the **Project Options** page with the selected option is displayed.

3.2.11.4.7 iLink 32

Topics

Name	Description
Linker Linking (↗ see page 878)	**Project ▶ Options ▶ Linker ▶ Linking** Use this dialog box to set Linker Linking options.
Linker (↗ see page 880)	**Project ▶ Options ▶ Linker** This is the top-level node of the Linker command line options. **Note:** Options marked with an asterisk (*) on the options pages are the default values.
Linker Output Options (↗ see page 881)	**Project ▶ Options ▶ Linker ▶ Output options** Use this dialog box to set Linker Output Setting options.
Linker Warnings (↗ see page 884)	**Project ▶ Options ▶ Linker ▶ Warnings** Use this dialog box to set Linker Warning options.

3.2.11.4.7.1 Linker Linking

Project ▶ Options ▶ Linker ▶ Linking

Use this dialog box to set Linker Linking options.

Build Configuration option	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

Linking options	Description
Dynamic RTL	Controls whether C RTL links statically or dynamically with runtime library (cc3280.dll). Default = false

Full debug information (-v)	Includes information in the output file needed to debug your application with the C++Builder integrated debugger or Turbo Debugger.
	On the command line this option causes the linker to include debugging information in the executable file for all object modules that contain debugging information. You can use the -v+ and -v- options to selectively enable or disable debugging information on a module-by-module basis (but not on the same command line where you use -v). For example, the following command includes debugging information for modules mod2 and mod3, but not for mod1 and mod4:
	`ILINK32 mod1 —v+ mod2 mod3 -v- mod4`
	Default = false

Keep output files (-Gk)	Tells the linker to keep output files that would otherwise be deleted on errors. The linker has been changed to delete its output file (EXE/DLL) if there are errors in the link. The old behavior was to leave these files and not delete them. Default = false
Maximum errors (-Enn)	Sets the specified number of errors encountered (nn) before the link is aborted. Default = 0
Generate package library (-Gl)	Generates a static package library containing code from all the .OBJs in the package so that it can be linked to. Default = false
Generate .drc file (-GD)	ILink32 generates Delphi compatible .RC files (drc files). The drc file has the same name as the executable file and is emitted to the same directory as the executable file. Default = false
Generate import library (-Gi)	Generates import library (DLL and package projects only). Default = false
Disable incremental link (-Gn)	Suppresses the generation of linker state files, disabling incremental linking. If you use -Gn, subsequent links take just as long as the first one. Default = false

Advanced options	**Description**
Do image checksum (-Gz)	Calculates the checksum of the target and places the result in the target's PE header. This is used for NT Kernel mode drivers and system DLLs. Default = false
Fast TLS (-Gt)	Allocate TLS (thread-local storage) from Windows instead of using the mechanism to share TLS. Default = false
Replace resources (-Rr)	Add and/or replace resources without stripping away the existing resources. Default = false
Case sensitive link (-c)	The linker differentiates between upper and lowercase characters in public and external symbols. Normally, this option should be set, since C and C++ are both case-sensitive languages. Default = true
Verbose (-r)	Sets the verbose option rlink32 and detailed information is emitted during the resource link. Default = false
Clear state before linking (-C)	Deletes the existing incremental linker state files and then recreates these files and continues with the link. This option allows you to refresh the state files. Default = false
File alignment (-Af)	Specifies page alignment for code and data within the executable file. The linker uses the file alignment value when it writes the various objects and sections (such as code and data) to the file. For example, if you use the default value of 0x200, the linker stores the section of the image on 512-byte boundaries within the executable file. When using this option, you must specify a file alignment value that is a power of 2, with the smallest value being 16. The old style of this option (/A:dd) is still supported for backward compatibility. With this option, the decimal number dd is multiplied by the power of 2 to calculate the file alignment value. The command-line version of this option (/Afxxxx) accepts either decimal or hexadecimal numbers as the file alignment value. The value setting is 512 (0x200). Default = 0x200

Object alignment (-Ao)	The linker uses the object alignment value to determine the virtual addresses of the various objects and sections (such as code and data) in your application. For example, if you specify an object alignment value of 8192, the linker aligns the virtual addresses of the sections in the image on 8192-byte (0x2000) boundaries. When using this option, you must specify an object alignment value that is a power of 2, with the smallest value being 4096 (0x1000) , the default. The command-line version of this option (/Ao) accepts either decimal or hexadecimal numbers as the object alignment value. Default = 0x1000
Delay load .DLLs (-d)	Delayed loading of DLLs is useful for DLLs that are used very infrequently by an application, but might have high startup costs. DLLs that have been delay loaded are not loaded and initialized until an entry point in the DLL is actually called. There is accompanying RTL support for delayed load DLLs that the user can hook into to handle errors on loading and to supplant the delayed load system, if so desired. Click [...] to display a dialog to manage of list of DLLs. Check the **Merge** box to act as if the immediate ancestor's DLLs are merged into this list, though this list is not actually changed.
Additional options	Enter any additional linker options.

General option	Description
Default	Saves the current settings as the default for each new project.

3.2.11.4.7.2 Linker

Project ▶ Options ▶ Linker

This is the top-level node of the Linker command line options.

Note: Options marked with an asterisk (*) on the options pages are the default values.

Build Configuration option	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

General option	Description
Default	Saves the current settings as the default for each new project.

Note: Many linker options and their switches are described in the other linker option pages for which links are provided below. You can also get information about Linker options by entering ilink32

in a command window:

```
  -C       Clear state before linking
  -wxxx    Warning control
```

```
    -Enn     Max number of errors
    -r       Verbose linking
    -q       Supress banner
    -c       Case sensitive linking
    -v       Full debug information
    -Gn      No state files
    -Gi      Generate import library
    -GD      Generate .DRC file
  Map File Control:
    -M       Map with mangled names
    -m       Map file with publics
    -s       Detailed segment map
    -x       No map
  Paths:
    -I       Intermediate output dir
    -L       Specify library search paths
    -j       Specify object search paths
  Image Control:
    -d       Delay load a .DLL
    -Af:nnnn Specify file alignment
    -Ao:nnnn Specify object alignment
    -ax      Specify application type
    -b:xxxx  Specify image base addr
    -Txx     Specify output file type
    -H:xxxx  Specify heap reserve size
    -Hc:xxxx Specify heap commit size
    -S:xxxx  Specify stack reserve size
    -Sc:xxxx Specify stack commit size
    -Vd.d    Specify Windows version
    -Dstring Set image description
    -Vd.d    Specify subsystem version
    -Ud.d    Specify image user version
    -GC      Specify image comment str
    -GF      Set image flags
    -Gl      Static package
    -Gpd     Design time only package
    -Gpr     Runtime only package
    -GS      Set section flags
    -Gt      Fast TLS
    -Gz      Do image checksum
    -Rr      Replace resources
```

See Also

Linking (↗ see page 878)

Output options (↗ see page 881)

Warnings (↗ see page 884)

3.2.11.4.7.3 Linker Output Options

Project▶Options▶Linker ▶Output options

Use this dialog box to set Linker Output Setting options.

Build Configuration option	**Description**
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

Map File options	Description

Type	**Map file with segments** *
	Only include segments in the map file. Happens when none of –m, –s, or –x is specified. This is the default.
	Map file with publics (-m)
	Instructs the linker to produce a map file that contains an overview of the application segments and two listings of the public symbols.
	The segments listing has a line for each segment showing the segment starting address, segment length, segment name, and the segment class.
	The public symbols are broken down into two lists, the first showing the symbols in sorted alphabetically, the second showing the symbols in increasing address order. Symbols with absolute addresses are tagged **Abs**.
	A list of public symbols is useful when debugging: many debuggers use public symbols, which lets you refer to symbolic addresses while debugging.
	Detailed segment map (-s)
	Creates the most comprehensive map file by adding a detailed map of segments to the map file created with the Publics option (-m). The detailed list of segments contains the segment class, the segment name, the segment group, the segment module, and the segment ACBP information. If the same segment appears in more than one module, each module appears as a separate line.
	The ACBP field encodes the A (alignment), C (combination), and B (big) attributes into a set of four bit fields, as defined by Intel. ILINK32 uses only three of the fields: A, C, and B. The ACBP value in the map is printed in hexadecimal. The following field values must be ORed together to arrive at the ACBP value printed.
	Field Value Description
	A (alignment) 00 An absolute segment
	20 A byte-aligned segment
	40 A word-aligned segment
	60 A paragraph-aligned segment
	80 A page-aligned segment
	A0 An unnamed absolute portion
	of storage
	C (combination) 00 Cannot be combined
	08 A public combining segment
	B (big) 00 Segment less than 64K
	02 Segment exactly 64K
	With the Segments options set, public symbols with no references are flagged idle. An idle symbol is a publicly defined symbol in a module that was not referenced by an EXTDEF record or by any other module included in the link. For example, this fragment from the public symbol section of a map file indicates that symbols Symbol1 and Symbol3 are not referenced by the image being linked:
	`0002:00000874 Idle Symbol1`
	`0002:00000CE4 Symbol2`
	`0002:000000E7 Idle Symbol3`
	Do not generate map (-x)
	Turns off the creation of the default linker map file.
	By default, the linker generates a map file with that contains general segment information including a list of segments, the program start address, and any warning or error messages produced during the link. There is no switch for this setting. Use the -x option to suppress the creation of this default map file.

Mangle names (-M)	Prints the mangled C++ identifiers in the map file, not the full name. This can help you identify how names are mangled (mangled names are needed as input by some utilities).

Versioning items	Description
OS version (-V)	Specifies the Windows version ID on which you expect your application to be run. The linker sets the Subsystem version field in the .EXE header to the number you specify in the input box. You can also set the Windows version ID in the SUBSYSTEM portion of the module definition file (.DEF file) However, any version setting you specify in the IDE or on the command line overrides the setting in the .DEF file. When you use the -Vd.d command-line option, the linker sets the Windows version ID to the number specified by d.d. For example, if you specify -V4.0, the linker sets the Subsystem version field in the .EXE header to 4.0, which indicates a Windows 95 application.
User version (-U)	Specifies the version ID of your executable. The linker sets the user version field in the executable's header to the number you specify. When you use the -Ud.d command-line option, the linker sets the application version ID to the number specified by d.d. For example, if you specify -V4.0, the linker sets the user version field in the executable's header to 4.0.

Image description option	Description
Image description (-D)	Saves the specified description in the PE image.

Intermediate output option	Description
Intermediate output	Tells the linker to place intermediate output files in the directory specified. Click [...} to display a **Browse for Folder** dialog.

PE file options	Description
Base address (-B)	Specify image base address. Preserve relocation table. Value in hex or decimal on 0x200 or 512 byte boundaries. Default value = 0x00400000
Minimum stack size (-Sc)	Spcifies the size of the committed stack in hexadecimal or decimal. The minimum allowable value for this field is 4K (0x1000) and any value specified must be equal to or less than the Reserved Stack Size setting (-S). Specifying the committed stack size here overrides any STACKSIZE setting in a module definition file. Default value = 0x00002000
Maximum stack size (-S)	Specifies the size of the reserved stack in hexadecimal or decimal. The minimum allowable value for this field is 4K (0x1000). Specifying the reserved stack size here overrides any STACKSIZE setting in a module definition file. Default value = 0x00100000

Minimum heap size (-Hc)	Specifies the size of the committed heap in hexadecimal or decimal. The minimum allowable value for this field is 0 and any value specified must be equal to or less than the Reserved Heap Size setting (-H). Specifying the committed heap size here overrides any HEAPSIZE setting in a module definition file. Default value = 0x00001000
Maximum heap size (-H)	Specifies the size of the reserved heap in hexadecimal or decimal. The minimum allowable value for this field is 0. Specifying the reserved heap size here overrides any HEAPSIZE setting in a module definition file. Default value = 0x00100000
Section flags (-GS)	The -GS switch lets you add flags to a named image section. This switch adds the flags to the existing flags for a given section. There is no way to remove default flags from a section. Allowable flags are: E - Executable C - Contains Code I - Contains initialized data R - Section is readable W - Section is writable S - Section is shared D - Section is discardable K - Section must not be cached P - Section must not be paged Default value = No flags
Image flags (-GF)	The GF switch allows you to set several flags on the image. The following flags are supported: -GF:SWAPNET -GF:SWAPCD -GF:UNIPROCESSOR -GF:LARGEADDRESSAWARE -GF:AGGRESSIVE SWAPNET tells the OS to copy the image to a local swap file and run it from there if the image resides on a network drive. SWAPCD tells the OS to copy the image to a local swap file and run it from there if the image resides on removable media (for example, CD, floppy, USB memory stick). UNIPROCESSOR tells the OS that this application cannot run on a multiprocessor system. LARGEADDRESSAWARE tells the OS that the application understands addresses bigger than 4G. AGGRESSIVE permits the OS to aggressively trim the working set of an application when the application is idle. This is ideal for screen savers and other processes that wish to stay out of the way of main line processes as much as possible. Default value = None

General option	Description
Default	Saves the current settings as the default for each new project.

3.2.11.4.7.4 Linker Warnings

Project ▶ Options ▶ Linker ▶ Warnings

Use this dialog box to set Linker Warning options.

Build Configuration option	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

Warnings options	Description
Enable all (-w)	Displays all warning and error messages.
Disable all (-w-)	Disables all warning and error messages
Selected	Chooses specific warnings to enable. This is the default. Click **Select All** to display all warnings in the list. Click **Clear All** to clear all selected warnings in the list Click **Defaults** to display the default warnings in the list

General option	Description
Default	Saves the current settings as the default for each new project.

3.2.11.4.8 List Editor

Project ▶ Options ▶ various paths

Use this dialog box to edit a list of semicolon-delimited strings.

Note: Not all of the options described below are available for all types of projects.

Item	Description
String list	Lists the strings currently set in the active build configuration.
Text field	Specifies a string to add or replace in the **String list**. Shows the currently selected string.
[Up Arrow]	Moves the selected string up in the list.
[Down Arrow]	Moves the selected string down in the list.
Ellipsis	Displays the **Browse for Folder** dialog box. Use this dialog box to specify a path.
Replace	Replaces the selected string with the text field content.
Add	Adds the string in the text field to the list.
Delete	Deletes the selected string.
Delete Invalid Paths	Deletes all invalid paths from the string list.
Inherited values	Lists the strings inherited from **All Configurations**. You cannot modify this list.

Inherit values from lower level settings	Causes the active build configuration to inherit the strings specified in **All Configurations**.

3.2.11.4.9 Implib

Topics

Name	Description
Librarian (↗ see page 886)	**Projects ▶ Options ▶ TLib** Use this dialog box to set Librarian (TLib) options. **Note:** Options marked with an asterisk (*) on the options pages are the default values.

3.2.11.4.9.1 Librarian

Projects ▶ Options ▶ TLib

Use this dialog box to set Librarian (TLib) options.

Note: Options marked with an asterisk (*) on the options pages are the default values.

Build Configuration option	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

TLib options	Description
Dynamic RTL	Controls whether C RTL links statically or dynamically with runtime library (cc3280.dll). Default = false
Case sensitive library (/c)	Warnings on case sensitive symbols. Default = false
Create extended directory (/E)	Creates extended directory. Default = false
Purge Comment Records (/0)	Purges comment records. Removes extra records such as line numbers. Default = false
Page size (/P)	Sets library page size. Default = 0x0010
Listing filename	Set listing filename. Click [...] to display a file selection dialog.

General option	Description
Default	Saves the current settings as the default for each new project.

3.2.11.4.10 Paths and Defines

Project ▶ Options ▶ Paths and Defines

Use this dialog box to set project paths and defines.

Build Configuration options	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

Paths options	Description
Include path: (-I)	Specifies the directories to be searched for include files. This is a set of include paths that are appended to any tool-specific include paths for the project as a whole. Standard include files are those you specify in angle brackets (<>) in an #include statement (for example, #include <myfile>). Click [...] to display a dialog allowing you to edit a list of paths. Check the **Merge** box to act as if the immediate ancestor's paths are merged into this list, though this list is not actually changed.

Library Path: (-L)	Specifies the directories the linker searches if there is no explicit path given for an .LIB module in the compile/link statement.
	The Specify Library Search Path uses the following command-line syntax:
	`/L<PathSpec>[;<PathSpec>][..]`
	The linker uses the specified library search path(s) if there is no explicit path given for the .LIB file and the linker cannot find the library file in the current directory. For example, the command
	`ILINK32 /Lc:\mylibs;.\libs splash.\common\logo,,,utils logolib`
	directs the linker to first search the current directory for SPLASH.LIB. If it is not found in he current directory, the linker then searches for the file in the C:\MYLIBS directory, and then in the .\LIBs directory. However, notice that the linker does not use the library search paths to find the file LOGO.LIB because an explicit path was given for this file.
	Click [...] to display a dialog to manage a list of paths. Check the **Merge** box to act as if the immediate ancestor's paths are merged into this list, though this list is not actually changed.

Intermediate Output:	Tells the linker to place intermediate output files in the directory specified. Also tells the compilers (dcc, bcc, tasm, brcc) where to put their compiled output; these are normally .obj and ..rcs files. Currently files that qualify for this placement are the linker state files. The .map file and .tds files go to the same directory as the output image, unless otherwise specified for the .map file. Click [...] to display a directory selection dialog.
Final Output:	Designates the directory where the final output of the build, such as the executable or DLL, is put. Click [...] to display a directory selection dialog.
BPI/Lib output: (-l)	Tells the linker to place bpi/lib output files in the directory specified, if they are generated. Click [...] to display a directory selection dialog.

Conditional Defines options	**Description**
Conditional Defines	Defines the specified identifier name to the null string. -D*name=string* defines name to string. In this assignment, string cannot contain spaces or tabs. You can also define multiple **#define** options on the command line using either of the following methods: This option applies to the entire project: all these defines are appended to those for specific compilers. Include multiple definitions after a single -D option by separating each define with a semicolon (;) and assigning values with an equal sign (=). For example: `BCC32.EXE -Dxxx;yyy=1;zzz=NO MYFILE.C` Include multiple -D options, separating each with a space. For example: `BCC32.EXE -Dxxx -Dyyy=1 -Dzzz=NO MYFILE.C` Click [...] to display a dialog to manage a list of defines.Check the **Merge** box to act as if the immediate ancestor's defines are merged into this list, though this list is not actually changed.

General option	**Description**
Default	Saves the current settings as the default for each new project.

3.2.11.4.11 Project Properties

Project or Tools ▶ Options ▶ Project Properties or Environment Options\C++ Options

Use this dialog box to set Project Properties that control certain aspects of how the project is managed in the IDE.

Note that this dialog can be displayed from either **Project Options** or **Tools Options**. If options are set in the dialog from **Project Options**, they apply only to that project. If options are set in the dialog from **Tools Options**, they apply to new projects.

C++ Project Properties options	**Description**
Manage include and library paths	If this option is checked, when a user adds files to the project, adds the paths for these files to the appropriate include path options to ensure the compiler/linker can find the files. If unchecked, does not update include paths automatically and the user takes responsibility for ensuring includes and library paths are correct. Default = true
Verify package imports and libraries	If checked, verifies that all package-related libraries can be found before linking. If a file is not found, displays a dialog asking the user for the location and update the include paths accordingly. If unchecked, don't perform this verification. Default = true
Show header dependencies in project manager	If checked, creates and shows a list of all header files on which a C/C++ file depends in the Project Manager if the information is available. If unchecked, does not generate the list. Default = false
Use auto-dependency checking when available	If an object file already exists for a source file, a tool creates a new object file if the modification date of the source is newer than that of the object file. If this option is checked, the tool builds a new object file if any of the include files on which a source file depends have a newer modification date than the object file. If unchecked, the tool does not check all include files. Enabling this option helps guarantee more accurate builds. Default = true
Show general messages	Show all messages from tools without filtering. Default = false

General option	Description
Default	Saves the current settings as the default for each new project.

3.2.11.4.12 tasm32

Topics

Name	Description
Turbo Assembler (see page 889)	Project ▶ Options ▶ Tasm This is the top-level node of the Turbo Assembler command line options. **Note:** Options marked with an asterisk (*) on the options pages are the default values.
Turbo Assembler Options (see page 889)	Project ▶ Options ▶ Tasm ▶ Options Use this dialog box to set Assembler Options.
Turbo Assembler Paths and Defines (see page 891)	Project ▶ Options ▶ Tasm ▶ Paths and Defines Use this dialog box to set Assembler Path and Define options.
Turbo Assembler Warnings (see page 892)	Project ▶ Options ▶ Tasm ▶ Warnings Use this dialog box to set Assembler Warnings options.

3.2.11.4.12.1 Turbo Assembler

Project ▶ Options ▶ Tasm

This is the top-level node of the Turbo Assembler command line options.

Note: Options marked with an asterisk (*) on the options pages are the default values.

Build Configuration option	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

General option	Description
Default	Saves the current settings as the default for each new project.

See Also

Paths and Defines (see page 891)

Options (see page 889)

Warnings (see page 892)

3.2.11.4.12.2 Turbo Assembler Options

Project ▶ Options ▶ Tasm ▶ Options

Use this dialog box to set Assembler Options.

Build Configuration option	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

Debugging options	Description
Debugging	**Full** (/zi) Lets you use all the features of the debugger to step through your program and examine or change data items. **Line numbers only** (/zd)* Tells the Turbo Assembler to include line-number records to synchronize source code display and data type information. This is the default. **None** (/zn) Disables debug information in the object file.

Code generation options	Description
Overlay	**Standard (no overlays)** (/os)* Creates standard object code without overlays. This is the default. **Standard (TLINK overlays)** (/o) Creates standard object code with TLINK overlays. **Phar Lap fixups** (/op) Creates object code with Phar Lap overlay-compatible fixups. **IBM fixups** (/oi) Creates object code with IBM overlay-compatible fixups.
Segment ordering	**Alphabetic** (/a) Orders the segments in alphabetic order. **Sequential** (/s)* Orders in the segments in the order in which they are encountered. This is the default.
Floating point	**Emulated** (/e)* Enable emulated floating-point instructions. This is the default. **Real** (/r) Enable real floating-point instructions.
Case sensitivity	**Case insensitive** (/mu)* Disables case sensitivity; treats symbol names as case-insensitive. This is the default. **Case sensitive** (/ml) Treats as uppercase all symbols used within the source file. **Globals case sensitive** (/mx) Treats only external and public symbols as case-sensitive.

General options	Description
Symbol table size (/kh)	Sets the maximum number of symbols an assembler file (.ASM) can use. The minimum allowable Hash table capacity is 8,192 bytes. The maximum allowable Hash table capacity is 32,768 bytes. Default = 8192
Maximum symbol length (/mv)	Sets the maximum length of symbols that Tasm can distinguish between. The minimum number allowed is 12. Default = 12
Maximum passes (/m)	Sets the maximum number of assembly passes. This is useful if you want the assembler to remove NOP instructions that were added because of forward references. Default = 1
Version id (/u)	Sets version emulation to specified version number. Default = 0
Impure code check (/p)	Checks for code segment overrides in protected mode. Default = false
Suppress .obj records (/q)	Suppresses .obj records not needed for linking. Default = false
Suppress messages (/t)	Suppresses messages if successful. Default = false
Display source lines in messages (/z)	Displays source line with error messages.

Assembler Directives option	Description
Assembler Directives (/j)	Defines an assembler startup directive (e.g., jIDEAL). This directive is assembled before the first line of the source file. Click [...] to display a dialog to manage a list of assembler directives. Check the **Merge** box to act as if the immediate ancestor's directives are merged into this list, though this list is not actually changed. Default = No directives specified.

General option	Description
Default	Saves the current settings as the default for each new project.

3.2.11.4.12.3 Turbo Assembler Paths and Defines

Project ▶ Options ▶ Tasm ▶ Paths and Defines

Use this dialog box to set Assembler Path and Define options.

Build Configuration option	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

3

Paths and Defines option	Description
Include path: (/i)	Specifies the drive and/or directories that contain program include files. Click [...] to display a **Include file search path** dialog to manage a list of paths. Check the **Merge** box to act as if the immediate ancestor's paths are merged into this list, though this list is not actually changed.
Defines: (/d)	Defines a list of defined symbols of the form name=value. Click [...] to display a **Define symbols** dialog to manage a list of defined symbols. Check the **Merge** box to act as if the immediate ancestor's defines are merged into this list, though this list is not actually changed.
.obj output directory	Sets the .obj output directory to the specified directory. Click [...] to display a directory selection dialog.

General item	Description
Default	Saves the current settings as the default for each new project.

General option	Description
Default	Saves the current settings as the default for each new project.

3.2.11.4.12.4 Turbo Assembler Warnings

Project ▶ Options ▶ Tasm ▶ Warnings

Use this dialog box to set Assembler Warnings options.

Build Configuration option	Description
Build Configuration	Displays the active build configuration. Use the drop-down menu to select another build configuration.
Save As...	Displays the **Save As** dialog box to save the current configuration's options to a file that can be loaded as a named option set.
Load...	Displays the **Apply Option Set** dialog box to apply the options in a named option set to the current configuration.

Warnings options	Description
Enable All (/w+)	Generates all warnings.
Disable All (/w-)	Disables all warnings.
Selected *	Chooses specific warnings to enable. This is the default. Click **Select All** to display all warnings in the list. Click **Clear All** to clear all selected warnings in the list Click **Defaults** to display the default warnings in the list

Note: Default is Selected

.

General option	Description
Default	Saves the current settings as the default for each new project.

3.2.11.4.13 Unavailable Options

Project ▶ Options

Some project options are no longer available in C++ Builder 2007. They may be available by using the tool switches.

For reference, they are listed here by major topics: C++ Compiler, Resource Compiler. Pascal Compiler, IDL to C++ Compiler, Linker, Librarian, and Turbo Assembler.

C++ Compiler: CodeGuard compile support	Description
CodeGuard debug level	**None** * CodeGuard is off. This is the default. **Level 0** (-vG0) Enables CodeGuard level 0. **Level 1** (-vG1) Enables CodeGuard level 1. This turns on the -vGd option. **Level 2** (-vG2) Enables CodeGuard level 2. This turns on the -vGd and -vGt options. **Level 3** (-vG3) Enables CodeGuard level 3. This turns on the -vGd, -vGc , and -vGt options.

C++ Compiler: Compatibility options	Description
Place no restrictions on where member pointers can point (-Vmv)	When this option is enabled, the compiler places no restrictions on where member pointers can point. Member pointers use the most general (but not always the most efficient) representation.

C++ Compiler: Paths and defines option	Description
Undefine any previous definitions of name (-U)	Undefines the previous definition of the specified identifier

C++ Compiler: Other Options	Description
Ignore system header files while generating dependency info (-mm)	Ignores system header files while generating dependency information
Console application (-tC)	

C++ Compiler: Target Setting option	Description
Windows application (-tW)	Target is a Windows application (same as -W)
Console application (-tWC)	Target is a console application (same as -WC)
Dynamic-link library (-tWD)	Generate a .DLL executable (same as -WD)
32–bit multi-threaded project (-tWM)	The compiler creates a multi-threaded .EXE or .DLL. (The command-line option -WM is supported for backward compatibility only; it has the same functionality as -tWM.) This option is not needed if you include a module definition file in your compile and link commands which specifies the type of 32-bit application you intend to build.
Generate a Unicode application (-tWU)	Generates a Unicode application

C++ Compiler: Assembler option	Description
Compile to .ASM (-S), then assemble to .OBJ (-B)	Causes the compiler to first generate an .ASM file from your C++ (or C) source code (same as the -S command-line option). The compiler then calls TASM32 (or the assembler specified with the -E option) to create an .OBJ file from the .ASM file. The .ASM file is then deleted. Your program will fail to compile with the -B option if your C or C++ source code declares static global variables that are keywords in assembly. This is because the compiler does not precede static global variables with an underscore (as it does other variables), and the assembly keywords will generate errors when the code is assembled.
Specify which assembler to use (-E)	Assemble instructions using the specified filename as the assembler. The 32-bit compiler uses TASM32 as the default assembler.
Specify assembler option, e.g. (-Tx) (-T)	Passes the specified option(s) to the assembler you specify with the -E option.

C++ Compiler: Batch Compile Support options	Description
Specify Stop batch compilation after n warnings (Default = 255) (-g)	Warnings: Stop After causes compilation to stop after the specified number of warnings has been detected. You can enter any number from 0 to 255. Entering 0 causes compilation to continue until either the end of the file or the error limit set in Stop after n errors has been reached, whichever comes first.
Specify Stop batch compilation after n errors (Default = None) (-j)	Errors: Stop After causes compilation to stop after the specified number of errors has been detected. You can enter any number from 0 to 255. Entering 0 causes compilation to continue until the end of the file or the warning limit set in Stop after n warnings has been reached, whichever comes first.

Stop batch compilation after first file with errors (-jb)	Aborts batch compilations after the first file that causes errors. For example,
	`BCC32 –c -gb *.ccp`
	`BCC32 –c –gb file1.cpp file2.cpp`
	Without the –jb flag, batch compilations continue to the next scheduled file, even after an earlier file has caused a error.

C++ Compiler: Template options	Description
Generate definitions for all template instances and merge duplicates (-Jgd)*	The compiler generates public (global) definitions for all template instances. If more than one module generates the same template instance, the linker automatically merges duplicates to produce a single copy of the instance. To generate the instances, the compiler must have available the function body (in the case of a template function) or the bodies of member functions and definitions for static data members (in the case of a template class), typically in a header file. This is a convenient way of generating template instances.
Print out all requested instantiations, using C++ syntax (-Jgi)	Prints out all requested instantiations using C++ syntax
Generate external references for all template instances (-Jgx)	The compiler generates external references to all template instances. If you use this option, all template instances that need to be linked must have an explicit instantiation directive in at least one other module.

Resource Compiler: Other options	Description
Instruction filename (@)	Takes instructions from the specified command file.
Resource (.RC) to compile	Lists resource files to compile

Resource Compiler: Output Setting options	Description
Output (.RES) file (-fo)	Renames the output .RES file. (By default, BRCC32 creates the output .RES file with the same name as the input .RC file.)

Pascal Compiler: : Map File options	Description
Detailed map file (-GD)	Creates detailed map file.
Map file with publics (-GP)	Creates map file with publics.
Map file with segments (-GS)	Creates map file with segments.
No map file*	No Map file specified

Pascal Compiler: Paths and Defines options	Description
Object file search paths (-O)	Sets the object file search paths to the specified paths.
Resource file search paths (-R)	Sets resource file search paths to the specified paths.
Unit search paths (-U)	Sets the unit search paths to the specified paths.

Pascal Compiler: Other options	Description
Build all units (-B)	Builds all units.
Find error (-F)	Finds specified error.
Make modified units (-M)	Makes modified units.
Look for 8.3 names also (-P)	Looks for 8.3 names also
Quiet compile (-Q)	Performs a quiet compile
Compiler directives (-$)	Sets compiler directives to the specified directives.
Imported data (-$G+)*	Enables imported data.
Runtime type info (-$M+)	Enables runtime type info.
Byte sized enumerations (-$Z1)*	Enables byte sized enumerations.
Word sized enumerations (-$Z2)	Enables word sized enumerations.
Double-word sized enumerations (-$Z4)	Enables double-word sized enumerations.
Export symbols (-$ObjExportAll On)	Exports symbols.
Real-type compatibility (-$REALCOMPATIBILITY ON)	Enables real-type compatibility.

Pascal Compiler: Linker EXE and DLL Output options	Description
Console target (-CC)	Outputs to console target.
GUI target (-CG)*	Outputs to GUI target.
Debug information in output (-V)	Displays debug information in output.

Generate remote debug symbols (.rsm) (-VR)	Generates remote debug symbols.

Pascal Compiler: Linker Memory option	**Description**
Set image base address (-K)	Sets image base address to the specified address.

Pascal Compiler: Message option	**Description**
Output hint messages (-H)*	Outputs hint messages.
Output warning messages (-W)*	Outputs warning messages.

IDL to C++ Compiler	**Description**
General	The IDL to C++ Compiler is no longer in the product.

Linker: Linking Options	**Description**
Suppress banner (-q)	Suppresses the banner.
Display time spent on link (-t)	Displays the time spent on link

Linker: Output Settings options	**Description**
Exe file	The name you want given to the executable file (.EXE, or .DLL). If you don't specify an executable file name, ILINK32 derives the name of the executable by appending .EXE or .DLL to the first object file name listed. (The linker assumes or appends an .EXE extensions for executable files if no extension is present. It also assumes or appends a .DLL extension for dynamic link libraries if no extension is present.)
Map file	Is the name you want given to the map file. If you don't specify a name, the map file name is given the same as exefile (but with the .MAP extension). (The linker appends a .MAP extensions if no extension is present.)

Linker: Application Type option	**Description**
32–bit Windows application (-aa)	Generates a protected-mode executable that runs using the 32-bit Windows API.

Windows device driver (-ad)	The application type is set to NATIVE, and the image checksum is calculated and set.
Console application (-ap)	Generates a 32-bit protected-mode executable file that runs in console mode.

Linker: Input Setting option	**Description**
Object files	The .OBJ files you want linked. Specify the path if the files aren't in the current directory. (The linker appends an .OBJ extensions if no extension is present.)
Library files	The library files you want included at link time. Do not use commas to separate the libraries listed. If a file is not in the current directory or the search path then you must include the path in the link statement. (The linker appends a .LIB extension if no extension is present.) The order in which you list the libraries is very important; be sure to use the order defined in this list: 1. Code Guard libraries (if needed) 2. List any of your own user libraries, noting that if a function is defined more than once, the linker uses the first definition encountered 3. IMPORT32.LIB (if you’re creating an executable that uses the Windows API) 4. Math libraries 5. Runtime libraries
Resource files	A list of .RES files (compiled resource files) to bind to the executable. (The linker appends an .RES extension if no extension is present.)
Def file	The module definition file for a Windows executable. If you don't specify a module definition (.DEF) file and you have used the /Twe or /Twd option, the linker creates an application based on default settings. (The linker appends a .DEF extension if no extension is present.)

Linker: Other Options	**Description**
Specify image comment str (-GC)	Adds comment strings to the image. These strings are inserted into the image directly after the object table in the PE file header. You can specify more than one string.

Linker: Packages option	**Description**
Package base name (-GB)	Assigns a base name for the package
Static package (-GI)	Generates a static package
Design time only package (-Gpd)	Generates a design-time-only package. (If neither /Gpr nor /Gpd is used, the resulting package works at both design time and runtime.)
Runtime only package (-Gpr)	Generates a runtime-only package. (If neither /Gpr nor /Gpd is used, the resulting package works at both design time and runtime.)

Linker: Paths and Defines option	**Description**

Library search path (-L)	Specifies the directories the linker will search if there is no explicit path given for an .LIB module in the compile/link statement.
	The Specify Library Search Path uses the following command-line syntax:
	`/L<PathSpec>[;<PathSpec>][..]`
	The linker uses the specified library search path(s) if there is no explicit path given for the .LIB file and the linker cannot find the library file in the current directory. For example, the command
	`ILINK32 /Lc:\mylibs;.\libs splash.\common\logo,,,utils logolib`
	directs the linker to first search the current directory for SPLASH.LIB. If it is not found in he current directory, the linker then searches for the file in the C:\MYLIBS directory, and then in the .\LIBs directory. However, notice that the linker does not use the library search paths to find the file LOGO.LIB because an explicit path was given for this file.

Specify object search paths (-j)	Specifies the directories the linker will search if there is no explicit path given for an object module in the compile/link statement.
	The Specify Object Search Path uses the following command-line syntax:
	`\j<PathSpec>[;<PathSpec>][..]`
	The linker uses the specified object search path(s) if there is no explicit path given for the object file and the linker cannot find the object file in the current directory. For example, the command
	`ILINK32 /jc:\myobjs;.\objs splash.\common\logo,,,utils logolib`
	directs the linker to first search the current directory for SPLASH.OBJ. If it is not found in he current directory, the linker then searches for the file in the C:\MYOBJS directory, and then in the .\OBJs directory. However, notice that the linker does not use the object search paths to find the file LOGO.OBJ because an explicit path was given for this file

Linker: PE File options	**Description**
Specify image base address (preserve relocation table) (-b)	Spcifies an image base address for your executable or DLL. The load address of the first object in the application or library is set to the number you specify, if possible, and all successive objects are aligned on 64K linear address boundaries; internal fixups are ignored. However, if the module cannot be loaded using the specified address, the operating system reverts to its default setting and applies internal fixups.

Linker: Windows Application Type options	**Description**
Windows Dynamic-link Library (-Tpd)	The linker generates a 32-bit protected-mode Windows .DLL file.
Windows Executable (-Tpe)	The linker generates a 32-bit protected-mode Windows .EXE file.
C++ Builder Package (-Tpp)	The linker generates a package. This switch is included automatically in package makefiles.

Librarian: Other options	**Description**
Force imports (-f)	Force imports by name

Ignore WEP (-i)	Ignores WEP
Remove module extentions (-o)	Removes module extensions
No warnings (-w)	No warnings

Librarian: Input Setting option	**Description**
Sourcefile name	Assigns a name to the sourcefile

Librarian: Output Setting option	**Description**
Library name	Assigns a name to the library

Turbo Assembler: Listing File options	**Description**
Generate cross-reference in listing file (/c)	Enables cross-reference in listing file. Tasm adds the cross-reference information to the symbol table at the end of the listing file.
Generate expanded listing (/la)	Generates the normal listing and includes the subset of C/C++ that generated the .ASM file.
Suppress symbol tables in listing file (/n)	Suppresses the symbol table in the listing file.
Include false conditionals in listing file (/x)	Includes false conditionals in listing. If a conditional (such as #if, #ifndef, #ifdef) evaluates to False, this option causes the statements inside the conditional block to appear in the listing file.

Turbo Assembler: Output Setting options	**Description**
Output object filename	Assigns a filename for the output object file
Listing file filename	Assigns a filename for the listing file
Cross-reference file filename	Assigns a filename for the cross-reference file

3.2.11.5 .NET Assemblies

Project ▶ Add Reference

Adds a .NET assembly reference to the current project.

The **Add Reference** dialog box is also available in the **Project Manager** by right-clicking a **References** folder and choosing Add

Reference.

Item	Description
Assembly Name	The name of the assembly.
Version	The version of the assembly.
Path	The location of the assembly.
Add Reference	Adds the selected (highlighted) reference to the **New References** list.
Browse	Displays a dialog box allowing you to navigate to an assembly.
Remove	Removes the reference currently selected in the **New References** list from the list.
OK	If the **New References** list contains any references, they are added to the project when you click **OK**.

Tip: Click any column heading to sort the display.

3.2.11.6 Project References

Project ▶ Add Reference

Adds a reference to a project that produces an assembly (.dll), such as a Class Library or Control Library. The reference will be added to the current project.

The **Add Reference** dialog box is also available in the **Project Manager** by right-clicking a **References** folder and choosing Add Reference.

Item	Description
Project Name	The name of the project that produces an assembly. Only projects within the current project group are listed.
Project Directory	The location of the project.
Add Reference	Adds the selected (highlighted) reference to the **New References** list.
Browse	Displays a dialog box allowing you to navigate to an assembly.
Remove	Removes the reference currently selected in the **New References** list from the list.
OK	If the **New References** list contains any references, they are added to the project when you click **OK**.

Tip: Click any column heading to sort the display.

3.2.11.7 Add to Repository

Project ▶ Add to Repository

Saves a customized form or project template in the **Object Repository** for reuse in other projects. The saved forms and templates are then available in the **New Items** dialog box when you choose **File ▶ New ▶ Other**.

Item	Description
Category	Lists the current category names.
New Category	Adds a folder with a new category to the **Object Repository**.
Title	Gives the template a name.

Description	Gives a description of the template. The description appears when you choose **File ▶ New ▶ Other**, select the template in the **Object Repository**, right-click, and choose View Details from the context menu.
Author	Identifies the author of the application. Author information appears only when you choose **File ▶ New ▶ Other**, select the template, right-click, and choose View Details from the context menu.
Browse	Opens the **Select icon** dialog box where you can select an icon to represent the item in the **Object Repository**. You can use a bitmap of any size, but it will be cropped to 60 x 40 pixels.

Tip: You can specify the path where you want the product to look for the file (BorlandStudioRepository.xml) that describes where the Object Repository

templates are located. If you want to change the path, choose **Tools ▶ Options ▶ Environment Options** and enter the path in the **Directory** text box.

After you save a form or project as a template, you can edit its description, delete the template, or change its icon by choosing **Tools ▶ Repository** and clicking the **Edit** button.

3.2.11.8 UDDI Browser

Project ▶ Add Web Reference

Searches for services and providers in the UDDI services sites with WSDL described services. Search by name or browse through available categorization schemas.

Item	Description
Microsoft production	Launches Microsoft's Production site for existing Web Services.
Microsoft Test	Launches Microsoft's Test site for existing Web Services.
XMethods Most recent	Launches www.xmethods.com/ home page and provides the most recent listings of Web Services.
XMethods full	Launches a full service list of all available Web Services.
IBM Secure	Launches IBMs UDDI Browser and lets you search for UDDI Business registeries.
Web reference folder name	Displays the name of the web reference as defined in the active WSDL document.
Add Reference	Adds the reference displayed in the Web reference folder name text box to your project and generates a proxy unit for the web reference.

3.2.11.9 Change Package

Adds required units to your package. This dialog box appears when the **Package Editor** tries to compile a package and detects that the package cannot be built, or is incompatible with another package currently loaded by the IDE. This occurs because the package uses one or more units that are found in another package.

Item	Description
View Details	Displays a list of the units that are required to build the package.
OK	Adds the missing package(s) to the `requires` clause of the package you are editing.
Cancel	Leaves the package you are editing as is.

Note: If you click Cancel

and do not apply the changes, errors may occur when the package is loaded.

3.2.11.10 Project Dependencies

Project ▶ Dependencies

Creates project dependencies within a project group. From the list, choose the projects to build before building the selected project.

Item	Description
Project Name	Displays the projects in the project group except for the selected project.
Path	Displays the location of the project file.

3.2.11.11 Add Languages

Project ▶ Languages ▶ Add Language

Adds one or more language resource DLLs to a project. Follow the instructions on each wizard page.

See Also

Localizing Applications (see page 18)

Adding Languages to a Project (see page 169)

3.2.11.12 Remove Language

Project ▶ Languages ▶ Remove Language

Removes one or more languages from the project. Follow the instructions on each wizard page.

See Also

Localizing Applications (see page 18)

Adding Languages to a Project (see page 169)

3.2.11.13 Set Active Language

Project ▶ Languages ▶ Set Active Language

Determines which language module loads when you run your application in the IDE. Before changing the active language, make sure you have recompiled the satellite assembly for the language you want to use.

Select the desired language and click **Finish**.

See Also

Localizing Applications (see page 18)

Adding Languages to a Project (see page 169)

3.2.11.14 New Category Name

Tools ▶ Repository ▶ Edit button ▶ New Category button

Use this dialog box to assign a name to a new category in the **Object Repository**.

Item	Description
New Category Name	Specifies the name for the new category being added to the **Object Repository**.

3.2.11.15 Information

Project ▶ Information

Views the program compilation information and compilation status for your project.

Item	Description
Source compiled	Displays total number of lines compiled.
Code size	Displays total size of the executable or assembly without debug information.
Data size	Displays memory needed to store the global variables.
Initial stack size	Displays memory needed to store the local variables.
File size	Displays size of final output file.
Status	Displays whether your last compile succeeded or failed.

3.2.11.16 Project Page Options

Project ▶ Project Page Options

Specifies an HTML file in your project as the Project Page for recording a description of the project, and various other notes and information. This page is automatically displayed in the IDE when you open the project.

Item	Description
Name	Specifies the name of the Project Page. The drop down list shows all HTML files in your project.
Resource Folder	Specifies the folder for additional HTML files or images files referenced by the Project Page.

3.2.11.17 Remove from Project

Project ▶ Remove from Project

Removes one or more files from the current project.

Item	Description
File list	Select the file that you want to remove. To select multiple files, press the `CTRL` key while selecting the files.

Note: If you attempt to remove a file that has been modified during the current edit session, you will be prompted to save your changes. If you have not modified the file, it is removed without a confirmation prompt.

Warning: Remove the file from your project before deleting the file from disk so that the product can update the project file accordingly.

3.2.11.18 Options

Project Manager ▶Right-click a satellite assembly▶Options command

Sets the assembly linker (`al.exe`) options for the satellite assembly selected in the **Project Manager**.

Item	Description
Culture	Specifies the culture string to associate with the assembly. Select a culture from the drop-down list. Corresponds to the /culture option.
Company	Specifies a string for the Company field in the assembly. Corresponds to the /company option.
Configuration	Specifies a string for the Configuration field in the assembly. Corresponds to the /configuration option.
Copyright	Specifies a string for the Copyright field in the assembly.
Description	Specifies a string for the Description field in the assembly. Corresponds to the /description option.
Trademark	Specifies a string for the Trademark field in the assembly. Corresponds to the /trademark option.
Product	Specifies a string for the Product field in the assembly. Corresponds to the /product option.
Product Version	Specifies a string for the Product Version field in the assembly. Corresponds to the /productversion option.
Title	Specifies a string for the Title field in the assembly. Corresponds to the /title option.
File Version	Specifies a string for the File Version field in the assembly. Corresponds to the /fileversion option.
Evidence	Specifies the file to embed in the assembly with the resource name of Security.Evidence. Corresponds to the /evidence option.
Keyfile	Specifies the file that contains a key pair or public key to sign an assembly. The compiler inserts the public key in the assembly manifest and then signs the final assembly with the private key. Corresponds to the /keyfile option.
Keyname	Specifies a container for a key pair. This will sign the assembly with a strong name by inserting a public key into the assembly manifest. Corresponds to the /keyname option.

Tip: The attributes listed in the Options

box will be available for viewing with reflection. For more information about the assembly linker, satellite assemblies, and reflection, refer to the .NET Framework SDK online Help.

3.2.11.19 Select Icon

Selects a bitmap to represent your template in the **New Items** dialog box.

You can use a bitmap of any size, but it will be cropped to 60 x 40 pixels.

3.2.11.20 Web Deploy Options

Project ▶ Web Deploy Options

Configures a finished ActiveX control or ActiveForm for deployment to a Windows Web server.

Tip: Set these options before you compile the ActiveX project and deploy it by choosing Project->Web Deploy.

Project Page

Use this page to specify file locations, a URL, and CAB file compression and version information.

Item	Description
Target Dir	Specifies the location of the ActiveX library file as a path on the Web server. This can be a standard path name or a UNC path. Click the Browse button to navigate to a desired directory, for example, `C:\INETPUB\wwwroot`.
Target URL	Specifies the URL for the ActiveX library file. See your Web server documentation for information on how it specifies URLs, for example, `http://mymachine.borland.com/`.
HTML Dir	Specifies the location where the HTML file that contains a reference to the ActiveX control should be generated. This can be a standard path name or a UNC path. Click the Browse button to navigate to the desired directory, for example, `C:\INETPUB\wwwroot`.
Use CAB file compression	Compresses the ActiveX library and all required packages and additional files that do not specify otherwise. Cabinet compression stores files in a file library, which can decrease download time by up to 70 percent.
Include file version number	Includes the version information specified on the **VersionInfo** page of the **Project Options** dialog box.
Auto increment release number	Automatically increments the project’s release number every time you choose **Project ▶ Web Deploy**. This updates the value on the **VersionInfo** page of the **Project Options** dialog box.
Deploy required packages	Deploys all packages listed on the **Packages** page along with the project.
Deploy additional files	Deploys all files listed on the **Additional Files** page along with the project.

Packages Page

Use this page to indicate which packages must be deployed with your project and how they should be deployed. Each package can specify its own options, overriding the defaults on the **Project** page. Packages that ship with this product are code signed with the CodeGear signature.

Note: You must check Deploy required packages

on the **Project** page to include these files. Otherwise, these packages are not deployed and you will not be able to select packages in the packages list.

Item	Description
Packages used by this project	Lists the packages that are required by your ActiveX library project. Select a package in this list to modify its options.

Compress in a separate CAB	Creates a separate `.cab` file for the package.
Compress in project CAB	Includes the package in the project `.cab` file.
Use file VersionInfo	If the package includes a **VersionInfo** resource, the version information in that resource is added to the `.inf` file for the project.
Target URL	Specifies the URL for the package file. If this is blank, the Web browser assumes the file already exists on the client machine. If the client does not have the package, the download of the ActiveX library fails.
Target directory	Specifies the directory where the package should be written on the server. This can be a standard path name or a UNC path. If this is blank, it indicates that the file already exists and should not be overwritten.

Additional Files Page

Use this page to indicate files other than packages that must be deployed with your project and how they should be deployed. You can use this page to add files or to specify the options for any file, overriding the defaults on the **Project** page.

Note: You must check Deploy additional files

on the **Project** page to include these files. Otherwise, these files are not deployed, and you will not be able to add or select files in the files list.

Item	Description
Files associated with project	Lists the files (other than packages) that are required by your ActiveX library project. You can add files to the list by clicking the **Add** button. You can remove the selected file by clicking the **Remove** button. Select a file to modify its options.
Compress in a separate CAB	Creates a separate `.cab` file for the package.
Compress in project CAB	Includes the package in the project `.cab` file.
Use file VersionInfo	If the package includes a **VersionInfo** resource, the version information in that resource is added to the `.inf` file for the project.
Target URL	Specifies the URL for the package file. If this is blank, the Web browser assumes the file already exists on the client machine. If the client does not have the package, the download of the ActiveX library fails.
Target directory	Specifies the directory where the package should be written on the server. This can be a standard path name or a UNC path. If this is blank, it indicates that the file already exists and should not be overwritten.

3.2.11.21 Build All Projects

Project ▶ Build All Projects

Compiles all of the source code in the current project group, regardless of whether any source code has changed. Building is useful when you are unsure which files have changed, or if you have changed project or compiler options.

See Also

Project Manager (see page 1038)

Compiling (see page 2)

3.2.11.22 Build Project

Project ▶Build Project

Rebuilds all files in your current project regardless of whether they have changed. Building is useful when you are unsure which files have changed, or if you have changed project or compiler options.

See Also

Project Manager (↗ see page 1038)

Compiling (↗ see page 2)

3.2.11.23 Compile and Make All Projects

Project ▶Compile

Project ▶Make All Projects

Compile (for Delphi) and Make (for C++) compiles only those files that have changed since the last build, as well as any files that depend on them. Compiling or making does not execute the application (see **Run ▶Run**).

See Also

Project Manager (↗ see page 1038)

Compiling (↗ see page 2)

3.2.11.24 Add to Project

Project ▶Add to Project

Adds another source file to an already open project.

Item	Description
Look in	Specifies the location where you want to locate a file or folder.
Files	Displays the files in the current directory that match the wildcards in **File name** or the file type in **Files Of Type**. You can display a list of files (default) or you can show details for each file.
File name	Displays a default name for the file you want to add.
File of type	Displays the files of the specified type. Only those files in the current directory that are of the specified type appear in the **Files** list box.
Open	Click **Open** to add the file or open a folder.

Tip: Press F1

in any list box or column to display tooltips with more information.

See Also

Project Manager (↗ see page 1038)

3.2.11.25 Add New Project

Project ▶Add New Project

Adds new projects via the **New Items** dialog box .

See Also

Project Manager (see page 1038)

New Items (see page 781)

3.2.11.26 Clean Package

Project ▶Clean Package

Removes previously compiled files and leaves behind only the source files needed to build the project. Specifically , it cleans out any .dcu's, .bpl's, etc., that were generated.

3.2.11.27 Default Options

Project ▶Default Options

Opens the default **Project Options** dialog box for the specified project type: C++ Builder, Delphi for Win32, Delphi for .NET, C# Builder, and Basic Builder. This option is only available when there is not an open project.

After the **Project Options** dialog opens help is available from each page of the dialog. Click **Help** or press `F1`.

See Also

Setting Project Options (see page 162)

Setting IDE

Project Options (see page 842)

3.2.11.28 Options

Project ▶Options

Opens the **Project Options** dialog that manages application and compiler options for your project. Making changes to your project only affects the current project. However, you can also save your selections as the default settings for new projects.

See Also

Setting Project Options (see page 162)

Build Configurations Overview (Delphi) (see page 5)

Build Configurations Overview (C++) (see page 6)

Named Option Sets Overview (see page 7)

3.2.11.29 Syntax Check for Project

Project ▶Syntax Check for Project

Checks the active project for incorrect symbols. This is an efficient way to check and see if a large project will build correctly prior to compiling. Errors are reported in the **Compiling** dialog with details shown in the **Messages** pane.

3.2.11.30 Update Localized Projects

Project ▶Languages▶Update Localized Projects

Updates resource modules. When you add an additional resource, such as a button on a form, you must update your resource modules to reflect your changes. Build and save your project before you update the resources.

3.2.11.31 View Source

Project ▶View Source

Shows the source of the project file that manages the running of the application. Dephi and Delphi .NET show a .dpr file. C++ shows .cpp file.

3.2.12 Propeditors

Topics

Name	Description
Delete Templates (↗ see page 914)	**Menu Designer context menu ▶Delete Templates** Use this dialog box to remove predefined templates from the list in the **Insert Template** dialog.
Insert Template (↗ see page 915)	**Menu Designer context menu ▶Insert From Template** Use this dialog box to add a menu using a predefined template.
Select Menu (↗ see page 915)	**Menu Designer context menu ▶Select Menu** Use this dialog box to select from a list of menus associated with the form whose menu is currently open in the **Menu Designer**.
Browse dialog box (↗ see page 915)	• **Insert Object dialog box▶Browse (when Create From File is selected)** • **Change Icon dialog box▶Browse** The Browse dialog box has multiple uses, depending on where you opened it: • To load an existing file into an OLE container. The file you select must be associated with an application that can be used as an OLE server. • To select an icon to represent an OLE object on the form.

Change Icon dialog box (↗ see page 916)	Use the Change Icon dialog box to specify an icon and a label for the object you are placing on the form. To open the Change Icon dialog box: 1. On the Insert Object dialog box, check Display As Icon. 2. Click Change Icon.
Color editor (↗ see page 917)	Use the Color editor to specify or define a color for the selected component. Changes you make using the Color editor are reflected in the Color property for a component. To open the Color editor: 1. Select any component or the form. 2. Double-click the Value column for the Color property or one of the other properties that use the Color editor.
DDE Info dialog box (↗ see page 918)	Use the DDE Info dialog box to specify, at design time, a DDE server application and a topic for a DDE conversation. To open the DDE Info dialog box: 1. Place a DDEClientConv component on the form. 2. With the component selected, do one of the following: • Click the ellipsis button in the Value column for the DdeService property or DdeTopic property. • Double-click the Value column for the DdeService property or DdeTopic property.
Filter editor (↗ see page 918)	Use the Filter editor to define filters for the OpenDialog component and the SaveDialog component. These common dialog boxes use the value of Filters in the List Files Of Type combo box to display certain files in the Files list box. Use the Filter editor to edit the Filter property. To open the Filter editor: 1. Place an OpenDialog component or SaveDialog component on the form. 2. With that component selected, do one of the following: • Click the ellipsis button in the Value column for the Filters property. • Double-click the Value column for the Filters property.
Font editor (↗ see page 919)	Use the Font editor to specify, at design time, a font and other font attributes for the selected component or form. Changes you make using the Font editor are reflected in the Font property for a component.
Action Manager editor (↗ see page 919)	Use the Action Manager editor at design time to add actions to ActionBands menus and toolbars through a TActionManager component. To display the Action Manager editor, select the TActionManager object and double-click the component or right-click and select Customize.
Action List editor (↗ see page 921)	Use the Action List editor at design time to add actions to a TActionList component.
Add Page dialog box (↗ see page 922)	Use the Add Page dialog box to add notebook pages to either the Notebook component or the TabbedNotebook component. To open this dialog box, in the Notebook editor, click Add.
Collection Editor (↗ see page 922)	The Collection Editor is used to edit the items maintained by a collection object. A collection object is a descendant of TCollection. The Collection Editor displays information about the items in the collection, and allows you to add, remove, or rearrange the individual items. For some collections, additional buttons are provided to allow other manipulations of the list. The items displayed in the list window of the Collection Editor can be selected using the mouse. Once an item is selected, its properties and events can be set using the Object Inspector.
Edit Page dialog box (↗ see page 923)	Use the Edit Page dialog box to edit existing notebook pages from either the Notebook component or the TabbedNotebook component. To open this dialog box, in the Notebook editor, click Edit.

IconView Items editor (↗ see page 924)	Use the IconView Items editor at design time to add or delete the items displayed in an iconview component. You can add or delete new items, and you can set the caption and image index for each item using the IconView Items editor.
Image List Editor (↗ see page 925)	Use the ImageList Editor at design time to add bitmaps and icons to a TImageList component. While working in the image list editor, you can click Apply to save your current work without exiting the editor, or click OK to save your changes and exit the dialog. Using the Apply button is especially useful because once you exit the dialog, you can't make any more changes to the existing images. To display the ImageList editor select the TImageList object and double-click the component or right-click and select ImageList Editor.
ListView Items Editor (↗ see page 926)	Use the ListView Items editor at design time to add or delete the items displayed in a listview component. You can add or delete new items and sub-items, and you can set the caption and image index for each item in the ListView Items editor. To display the ListView Items editor, select the TListView object and double-click the Items property value in the Object Inspector.
New Standard Action Classes dialog box (↗ see page 927)	Use the New Standard Action Classes dialog box to add a predefined action to your action list. Standard actions perform common tasks such as navigating datasets, managing the windows in an MDI application, or working with the Windows clipboard. Each standard action performs a specific function when invoked, and enables or disables any linked controls as appropriate. Choose the action you want to add from the list and click OK. For a description of each predefined action class, see Predefined Action Classes.
String List editor (↗ see page 927)	Use the String List editor at design time to add, edit, load, and save strings into any property that has been declared as TStrings. To open the String List editor: 1. Place a component that uses a string list on the form. 2. With that component selected, do one of the following: • Click the ellipsis in the Value column for any property that has been declared as TStrings, such as the Items property of the ComboBox property. • Double-click the word (TStrings) in the Value column for any property that has been declared as TStrings. **Note:** If the property is a value list,... more (↗ see page 927)
TreeView Items Editor (↗ see page 927)	Use the TreeView Items editor at design time to add items to a tree view component, delete items from a tree view component, or load images from disk into a tree view component. You can specify the text associated with individual tree view items, and set the image index, selected index, and state index for items. To display the TreeView Items editor, select the TTreeView object and double-click the Items property value in the Object Inspector.
Value List editor (↗ see page 928)	Use the Value List editor at design time to add, edit, load, and save name-value pairs into any property that has been declared as TStrings. To open the Value List editor: 1. Place a component that uses a string list on the form. 2. With that component selected, do one of the following: • Click the ellipsis in the Value column for any property that has been declared as TStrings. • Double-click the word (TStrings) in the Value column for any property that has been declared as TStrings. To add items to the value list, type the name of the item in the Key... more (↗ see page 928)

Input Mask editor (↗ see page 929)	Use the Input Mask editor to define an edit box that limits the user to a specific format and accepts only valid characters. For example, in a data entry field for telephone numbers you might define an edit box that accepts only numeric input. If a user then tries to enter a letter in this edit box, your application will not accept it. Use the Input Mask editor to edit the EditMask property of the MaskEdit component.
Insert Object dialog box (↗ see page 930)	Use the Insert Object dialog box at design time to insert an OLE object into an OleContainer component. The OleContainer component enables you to create applications that can share data with an OLE server application. After you insert an OLE server object in your application, you can double-click the OleContainer component to start the server application. Select whether or not you want to create a new file using the associated OLE server or use an existing file. If you use an existing file, it must be associated with an application that can act as an OLE server.
Loading an image at design time (↗ see page 931)	Use the **Picture Editor** to load images onto any of several graphic-compatible components and to specify an image to represent a form when it is minimized at runtime. Each graphic-compatible component has a property that uses the **Picture Editor**. To load an image at design time: 1. Add a graphic-compatible component (such as TImage) to your form. 2. To automatically resize the component so that it fits the graphic, set the component's **AutoSize** property to true before you load the graphic. 3. In the **Object Inspector**, select the property that uses the **Picture Editor**. 4. Either double-click in the Value column,... more (↗ see page 931)
Load Picture dialog box (↗ see page 931)	Use the Load Picture dialog box to select an image to add to any of the graphic-compatible components and to specify an icon for your form. To open the Load Picture dialog box, in the Picture editor, click Load.
Load String List dialog box (↗ see page 932)	Use the **Load String List** dialog box to select a text file to load into a property of type TStrings. To open this dialog box: 1. Bring up the **String List Editor**. 2. Right-click and choose Load.
Masked Text editor (↗ see page 932)	Use the Mask Test editor to enter Values into the edit mask. Use the Masked Text editor to edit the Text property of the MaskEdit component. To open the Masked Text editor: 1. Place an MaskEdit component on the form. 2. With that component selected, do one of the following: • Click the ellipsis button in the Value column for the Text property. • Double-click the Value column for the Text property.

Notebook editor (see page 933)	Use the Notebook editor to add, edit, remove, or rearrange pages in either a TabbedNotebook component or Notebook component. You can also use the Notebook editor to add or edit Help context numbers for each notebook page. The Notebook editor displays the current pages of the notebook in their current order, and it also displays the Help context associated with that page. To open the Notebook editor: 1. Place a Notebook component or TabbedNotebook component from the Win 3.1 Component palette page on the form. 2. With that component selected, do one of the following: • Click the ellipsis button in the Value... more (see page 933)
Open dialog box (see page 933)	Use the Open dialog box at design time to load a multimedia file into the MediaPlayer component. To open the Open dialog box: 1. Place a MediaPlayer component on the form. 2. With that component selected, do one of the following: • Click the ellipsis button in the Value column for any of the properties listed below. • Double-click the Value column for either of the properties listed below.
Paste Special dialog box (see page 934)	Use the Paste Special dialog box to insert an object from the Windows clipboard into your OLE container.
Picture editor (see page 934)	Use the Picture editor to select an image to add to any of the graphic-compatible components and to specify an icon for your form. To open the Picture editor: 1. Place a graphic-compatible component (such as TImage) on the form. 2. With that component selected, do one of the following: 3. Click the ellipsis button in the Value column for properties (such as the Picture property of TImage) related to editing the picture. 4. Double-click the Value column for properties related to editing the picture. **Note:** To open the Picture editor from an Image component, you can also double-click the component on the form.... more (see page 934)
Save Picture As dialog box (see page 935)	Use the **Save Picture As** dialog box to store the image loaded in the **Picture Editor** into a new file or directory. To open the **Save Picture As** dialog box, in the **Picture Editor**, click Save As.
Save String List dialog box (see page 936)	Use the Save string list dialog box to store the string list from the String List editor into a text file. To open this dialog box: 1. Bring up the **String List Editor**. 2. Right-click and choose Save.

3.2.12.1 Delete Templates

Menu Designer context menu ▶Delete Templates

Use this dialog box to remove predefined templates from the list in the **Insert Template** dialog.

Edit Menu	Removes the standard Edit menu with items that include Undo, Repeat, Cut, Copy, Paste, Paste Special, Find, Replace, Go To, Links, and Objects.
File Menu	Removes the standard File menu with items that include New, Open, Save, Save As, Print, Print Setup, and Exit.
File Menu (for TextEdit Example)	Removes the File menu with the standard items, and also includes the Close item.
Help Menu	Removes the Help menu with items that include Contents, Search for Help On, How to Use Help, and About.
Help Menu (Expanded)	Removes the Help menu with the standard items, along with additional items that include Index, Commands, Procedures, Keyboard, and Tutorial.
MDI Frame Menu	Removes the set of menus that contain the same items as the File, Edit, Window, and Help (Expanded) menus.
Window	Removes the Window menu with items that include New Window, Tile, Cascade, Arrange All, Hide, and Show.

3.2.12.2 Insert Template

Menu Designer context menu ▶Insert From Template

Use this dialog box to add a menu using a predefined template.

Edit Menu	Adds a standard Edit menu with items that include Undo, Repeat, Cut, Copy, Paste, Paste Special, Find, Replace, Go To, Links, and Objects.
File Menu	Adds a standard File menu with items that include New, Open, Save, Save As, Print, Print Setup, and Exit.
File Menu (for TextEdit Example)	Adds a File menu with the standard items, and also includes the Close item.
Help Menu	Adds a Help menu with items that include Contents, Search for Help On, How to Use Help, and About.
Help Menu (Expanded)	Adds a Help menu with the standard items, along with additional items that include Index, Commands, Procedures, Keyboard, and Tutorial.
MDI Frame Menu	Adds a set of menus that contain the same items as the File, Edit, Window, and Help (Expanded) menus.
Window	Adds a Window menu with items that include New Window, Tile, Cascade, Arrange All, Hide, and Show.

3.2.12.3 Select Menu

Menu Designer context menu ▶Select Menu

Use this dialog box to select from a list of menus associated with the form whose menu is currently open in the **Menu Designer**.

3.2.12.4 Browse dialog box

- **Insert Object dialog box ▶Browse (when Create From File is selected)**
- **Change Icon dialog box ▶Browse**

The Browse dialog box has multiple uses, depending on where you opened it:

- To load an existing file into an OLE container. The file you select must be associated with an application that can be used as an OLE server.
- To select an icon to represent an OLE object on the form.

Item	Description
Source	Enter the name of the file you want to load or wildcards to use as filters in the Files list box.
Files	Displays the files in the current directory that match the wildcards in the Source edit box or the file type in the Files of Type combo box.
Files of Type	Choose the type of file you want to use as the OLE server. By default all files in the current directory are displayed in the Files list box.
Directories	Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the Source edit box or the file type in the Files of Type combo box appear in the Files list box.
Drives	Select the current drive. The directory structure for the current drive appears in the Directories list box.

3.2.12.5 Change Icon dialog box

Use the Change Icon dialog box to specify an icon and a label for the object you are placing on the form.

To open the Change Icon dialog box:

1. On the Insert Object dialog box, check Display As Icon.
2. Click Change Icon.

Item	Description
Icon Radio	Select the icon to use: **Current** uses the current icon. **Default** uses the default icon. **From File** enables you to specify an icon using a fully qualified path name. If you do not know the icon name or the path, click **Browse** to open the **Browse** dialog box. The display box below the edit box shows all the available icons in the specified file. To choose an icon, select it.
Label	Enter the label that is to appear below the icon on the form.

Browse	Click **Browse** to open the **Browse** dialog box, where you can select an icon to represent the inserted object on the form.
Sample Icon Display	Displays how the icon and label will appear on the form.

See Also

Browse (↗ see page 915)

3.2.12.6 Color editor

Use the Color editor to specify or define a color for the selected component. Changes you make using the Color editor are reflected in the Color property for a component.

To open the Color editor:

1. Select any component or the form.
2. Double-click the Value column for the Color property or one of the other properties that use the Color editor.

Basic colors grid

Displays selection of standard colors. Click a color to apply it to the selected component.

Custom colors grid

Displays the colors that you have created. You can create custom colors by clicking Define Custom Colors.

Define Custom Colors

Click Define Custom Colors to expand the Color editor to show options that enable you to create your own colors.

Color field

Displays the spectrum of available colors. The crosshairs indicate the current color.

Click anywhere or drag in the color field to select a color. When you select a color here and then click Add To Custom Colors, the selected color is added to one of the Custom Color boxes so you can use it again.

Color|Solid

Displays the currently selected color and its closest solid color. Double-click the solid color to make it the current color.

Hue

Enter a value for the hue. Hue is the "actual" color, for example, red, yellow-green, or purple. Hue refers to the color without regard to saturation or brightness (luminosity).

Sat(uration)

Enter a value for the saturation. Saturation refers to how much gray is in the color. The Sat(uration) field shows the saturation from 0 (medium gray) to 240 (pure color).

Note: Saturation affects how clear the color is. Luminosity affects how bright the color is.

Lum(inosity) and the Slider Control

Enter a value for the luminosity, or drag the pointer on the slider to set the luminosity. Luminosity is the brightness of a color. The Lum(inosity) field shows the luminosity from 0 (black) to 240 (white). The column to the right of the color field shows the range of luminosity for the current color. Slide the arrow to the right of the column up or down to adjust the luminosity. When you change the luminosity, the Red/Green/Blue color values also change accordingly.

Red/Green/Blue

Enter values for the proportion of red, green, and blue you want in your color. The values in these fields show the balance of red, green, and blue in the current color. This is sometimes called the RGB color. The range of available values for an RGB color is 0 to 255.

Add To Custom Colors

Click to add the color you have defined to the Custom Color grid on the Color editor.

3.2.12.7 DDE Info dialog box

Use the DDE Info dialog box to specify, at design time, a DDE server application and a topic for a DDE conversation.

To open the DDE Info dialog box:

1. Place a DDEClientConv component on the form.
2. With the component selected, do one of the following:

- Click the ellipsis button in the Value column for the DdeService property or DdeTopic property.
- Double-click the Value column for the DdeService property or DdeTopic property.

DDE Info options

The following options are available in the DDE Info dialog box:

Dde Service

Specify the server application for the DDE conversation. The application you specify is entered into the Value column for the DdeService property.

You do not need to specify an extension for the server application.

If the directory containing the application is not on your path, you need to specify a fully qualified path.

Dde Topic

Enter the topic for a DDE conversation. The topic is a unit of data, identifiable to the server, containing the linked text. For example, the topic could be the file name of a spreadsheet.

When the server is a VCL-based application, the topic is the name of the form containing the data you want to link.

If the directory containing the topic is not on your path, you need to specify a fully qualified path.

Paste Link

Click Paste Link to paste the application name and file name from the contents of the Clipboard into the App and File edit boxes.

This button is active only when the clipboard contains data from an application that can be a DDE server.

3.2.12.8 Filter editor

Use the Filter editor to define filters for the OpenDialog component and the SaveDialog component. These common dialog boxes use the value of Filters in the List Files Of Type combo box to display certain files in the Files list box.

Use the Filter editor to edit the Filter property.

To open the Filter editor:

1. Place an OpenDialog component or SaveDialog component on the form.
2. With that component selected, do one of the following:

- Click the ellipsis button in the Value column for the Filters property.
- Double-click the Value column for the Filters property.

Filter Name column

Enter the name of the filter you want to appear in the Files of Type combo box.

Filter column

Enter wildcards and extensions that will define your filter. For example, *.txt would display only files with the .txt extension.

To apply multiple file extensions to your filter, separate them using a semicolon.

3.2.12.9 Font editor

Use the Font editor to specify, at design time, a font and other font attributes for the selected component or form. Changes you make using the Font editor are reflected in the Font property for a component.

Opening the Font editor

To open the Font editor:

1. Select any component or the form.
2. Do one of the following:

- Click the ellipsis button in the Value column for the Font property or one of the other properties that use the Font editor.
- Double-click the Value column for the Font property or one of the other properties that use the Font editor.

Font

Select a font from the list of all the available fonts you can use in your application.

Font style

Select a style for the font. This combo box displays only those styles that are available for the selected font. For most of the available fonts, there are four possible styles:

- Regular
- Italic
- **Bold**
- **Bold Italic**

Size

Select a size for the font (in points). This combo box displays only those font sizes that are available for the selected font.

Effects

Check these options to make the text strike-through or underlined.

Color

Select a color for the font. This combo box lists all the available colors for the selected font.

Sample area

Displays a sample of the selected font before you apply it to the components. The font within this area is updated with every change you make to the font settings.

Script

Lists the available language scripts for the selected font.

3.2.12.10 Action Manager editor

Use the Action Manager editor at design time to add actions to ActionBands menus and toolbars through a TActionManager component.

To display the Action Manager editor, select the TActionManager object and double-click the component or right-click and select Customize.

Toolbars tab

The toolbars tab allows you to quickly add toolbars (of type TActionToolBar) to your application by pressing the New button. Use the Delete button to remove unwanted toolbars. All of the ActionBands toolbars in the application are listed in the Toolbars box. Check or uncheck them to make them visible or invisible. You can change the caption options for the toolbars by changing the Caption Options in the Toolbar Options box.

Actions tab

At the top right corner of the actions tab is a toolbar containing four buttons. These are as follows:

Button When clicked

New Action Inserts a new action into the list. By clicking the drop-down arrow next to the button, you can choose whether to add a new action that you define, or a standard (predefined) action. If you choose Standard Action, you will be presented with a dialog where you can choose the predefined action.

Delete Deletes the action currently selected in the list boxes.

Arrow buttons Moves the currently selected action up or down to change its position in the list.

The lower portion of the actions tab contains two list boxes that represent the current list of actions. The first list indicates the value of the Category property of the action. You can change this value by selecting the action and changing the value of the Category property in the Object Inspector.

The second list indicates the name of the action. You can change this value by selecting the action and changing the value of the Name property in the Object Inspector.

Right click in the lower portion of the Action Manager editor to display the ActionManager context menu. This contains the following items:

Command When checked

New Action Adds a new (not predefined) action to the Action Manager editor. You can then use the Object Inspector to edit its properties.

New Standard Action Displays the Standard Actions dialog box, where you can select a predefined action.

Move Up Moves the currently selected action toward the beginning of the list.

Move Down Moves the currently selected action toward the end of the list.

Cut Cuts the currently selected action to the clipboard, removing it from the list.

Copy Copies the currently selected action to the clipboard without removing it from the list.

Paste Pastes an action from the clipboard above the currently selected action.

Delete Deletes the currently selected action.

Select All Selects all actions in the list.

Options tab

This tab has two sections. The top section, Personalized Menus and Toolbars, has a check box (marked "Menus show recently used items first") that dictates how menu items will be shown. There is also a button, marked "Reset Usage Data," which restores the action bands of the application to their initial settings.

The Other section of the Options tab contains a check box which causes large icons to appear on action bands. Another check box causes tips to appear on toolbars. A third check box will (if tips are selected) show shortcut keys in the tips. Finally, there is a field that allows you choose what type of animation will be used when menus open.

3.2.12.11 Action List editor

Use the Action List editor at design time to add actions to a TActionList component.

Opening the Action List editor

To display the Action List editor, Select the TActionList object and double-click the component or right-click and select Action List editor.

Toolbar

At the top of the Action List editor is a toolbar containing four buttons. These are as follows:

Button

When clicked

New Action Inserts a new action into the list. By clicking the drop-down arrow next to the button, you can choose whether to add a new action that you define, or a standard (predefined) action. If you choose Standard Action, you will be presented with a dialog where you can choose the predefined action.

Delete Deletes the action currently selected in the list boxes.

Arrow buttons Moves the currently selected action up or down to change its position in the list.

Right click the Toolbar to display the ActionList toolbar context menu. This contains one item:

Command When clicked

Text labels Displays or hides the labels on the buttons in the toolbar.

List boxes

The lower portion of the Action List editor contains two list boxes that represent the current list of actions. The first list indicates the value of the Category property of the action. You can change this value by selecting the action and changing the value of the Category property in the Object Inspector.

The second list indicates the name of the action. You can change this value by selecting the action and changing the value of the Name property in the Object Inspector.

Right click in the lower portion of the Action List editor to display the ActionList context menu. This contains the following items:

CommandWhen clicked

New Action Adds a new (not predefined) action to the Action List editor. You can then use the object Inspector to edit its properties.

New Standard Action Displays the Standard Actions dialog box, where you can select a predefined action.

Move Up Moves the currently selected action toward the beginning of the list.

Move Down Moves the currently selected action toward the end of the list.

Cut Cuts the currently selected action to the clipboard, removing it from the list.

Copy Copies the currently selected action to the clipboard without removing it from the list.

Paste Pastes an action from the clipboard above the currently selected action.

Delete Deletes the currently selected action.

Select All Selects all actions in the list.

Panel Descriptions Displays or hides labels over the listbox indicating their purpose.

Toolbar Displays or hides the toolbar.

3.2.12.12 Add Page dialog box

Use the Add Page dialog box to add notebook pages to either the Notebook component or the TabbedNotebook component.

To open this dialog box, in the Notebook editor, click Add.

The following options are available in the Add Page dialog box:

Page Name

Enter the name of the notebook page. There is a 255-character limit on page names.

Help Context

Enter the context ID number for the notebook page. This number is significant if you want to have context-sensitive Help for the individual pages of the notebook. The Help context is optional.

3.2.12.13 Collection Editor

The Collection Editor is used to edit the items maintained by a collection object. A collection object is a descendant of TCollection. The Collection Editor displays information about the items in the collection, and allows you to add, remove, or rearrange the individual items. For some collections, additional buttons are provided to allow other manipulations of the list.

The items displayed in the list window of the Collection Editor can be selected using the mouse. Once an item is selected, its properties and events can be set using the Object Inspector.

Opening the Collection editor

To display the Collection editor, first place the component that uses the collection on a form. Select the property that is implemented using the collection (listed in parentheses in the preceding table), and click on the ellipsis. For some components, the Collection Editor may also be displayed by right-clicking the component, and selecting the appropriate editor from the context menu.

Dialog box options

The following options are available in the Collection editor.

Item list

The Item list displays the properties listed in the third column of the preceding table for each item in the collection. The properties for a selected item are displayed in the Object Inspector and are edited there.

Add button

Adds a new item to the collection. You can select the item and edit its parameters in the Object Inspector.

Delete button

Removes the selected item from the collection.

Move Up/Down buttons

Change the order of the items. For most collections, the order determines the order in which items are displayed or used by the object that maintains the collection.

Add All Fields button (TDBGridColumns only)

Add a column for every field in the dataset to which the data-aware grid is bound. This button is only enabled if the data-aware grid is bound to an active dataset.

Restore Defaults button (not for all collections)

Restore the default properties (obtained from the field component) of the currently selected column. This button is enabled if the currently selected column is bound to a field (the FieldName property is set).

Read From Dictionary button (TCheckConstraints only)

Add a CheckConstraint object for every record-level constraint in the data dictionary. Each CheckConstraint object will have its ImportedConstraint property set to the constraint from the dictionary.

Examples of collection items

The following table provides examples of collection items. Note that the list is not complete as new collection items are being added all the time.

Collection	Item type	Properties displayed	Use
TAggregates	TAggregate	Aggregates	At design time, you can use the editor to add aggregate fields to a client dataset. When you define aggregate fields at design time, the editor automatically creates the TAggregate objects for them.
TCheckConstraints	TCheckConstraint	ImportedConstraint, or, if no ImportedConstraint is blank, CustomConstraint	Represents record-level constraints for the data in a dataset. (Constraints property)
TCoolBands	TCoolBand	Text	Represents a set of bands in a CoolBar component. (Bands property)
TDBGridColumns	TColumn	FieldName	Represents the field binding and display properties of a column in a data-aware grid. (Columns property)
TIndexDefs	TIndexDef	IndexDefs	Describes an index in a database table.
THeaderSections	THeaderSection	Text	Represents the display properties of the sections in a HeaderControl object. (Sections property)
TListColumns	TListColumn	Caption	Represents the columns of a report-style List View component. (Columns property)
TStatusPanels	TStatusPanel	Text	Represents the individual panels of a StatusBar component. (Panels property)
TWebActionItems	TWebActionItem	Name, PathInfo, Enabled, and Default	Represents the action items that create the responses to HTTP request messages for a Web dispatcher or Web module. (Actions property)

3.2.12.14 Edit Page dialog box

Use the Edit Page dialog box to edit existing notebook pages from either the Notebook component or the TabbedNotebook component.

To open this dialog box, in the Notebook editor, click Edit.

Edit Page options

The following options are available:

Page Name

Enter the name of the notebook page. There is a 255-character limit on page names.

Help Context

Enter the context ID number for the notebook page. This number is significant if you want to have context-sensitive help for the individual pages of the notebook. The Help context is optional.

3.2.12.15 IconView Items editor

Use the IconView Items editor at design time to add or delete the items displayed in an iconview component. You can add or delete new items, and you can set the caption and image index for each item using the IconView Items editor.

Opening the IconView Items editor

You can display the IconView Items editor in these ways:

- Select a TIconView object on a form and click the ellipsis next to the Items property value in the Object Inspector, or
- Double-click a TIconView object on a form, or
- Right-click a TIconView object on a form and choose Items Editor on the context menu.

Using the IconView Items editor

The IconView Items editor contains an Items group box with an Items list box, a New Item button and a Delete button. When you first add an iconview control to a form, the Items list box is empty and the Delete button as well as the Item properties on the right are disabled. When you enter or change item properties for a selected item, the Apply button is enabled so that you can activate changes immediately.

The IconView Items editor also contains an Item Properties group box for setting the properties of the iconview item currently selected in the Items list box. The Item Properties group box contains a Caption edit box and an Image Index edit box.

Items group box

Items list box Shows the iconview items. These are contained in the Items property of the iconview control.

New Items button Lets you add items to the iconview.

Delete button Lets you delete a selected item in the Items list box.

Create and delete iconview items and subitems in the Items group box. To create a new item, click New Item. A default item caption appears in the Items list box. Specify an item's properties, including its caption, in the Items Properties group box. When you create a new item, or select an existing item, the New SubItem button is enabled so that you can nest items within other items in the iconview. If the Items list box contains items, the Delete button is also enabled. To delete an item, select it in the Items list box and click Delete.

Item Properties group box

Caption Names the selected item in the Items list box. The name appear in the icon view.

Image Index Lets you specify the number of an image to be used next to the selected iconview item. These are contained in the Images property of the iconview control.

Set the properties for a selected item in the Item Properties group box. Enter a name for the item in the Caption edit box. As you enter the name, it changes in the Items list box.

To display an image to the left of an item that is not currently selected, specify the index number of the image in the Image Index edit box. To suppress image display, set Image Index to 0 (the default).

3.2.12.16 Image List Editor

Use the ImageList Editor at design time to add bitmaps and icons to a TImageList component.

While working in the image list editor, you can click Apply to save your current work without exiting the editor, or click OK to save your changes and exit the dialog. Using the Apply button is especially useful because once you exit the dialog, you can't make any more changes to the existing images.

To display the ImageList editor select the TImageList object and double-click the component or right-click and select ImageList Editor.

Selected Image

The selected image control displays the currently selected image. This image can be changed by clicking on another image in the Images list view below. When an image is selected, you can delete it from the list of images. If the image was not added to the image list before the current invocation of the editor, you can use the other controls to alter its properties. However, once the image list editor is closed, these properties are fixed and the selected image controls are grayed if the ImageList Editor is again displayed and that image is selected.

Transparent color

Use the Transparent color drop-down to specify which color is used to create a mask for drawing the image transparently. The default transparent color is the color of the bitmap's left-most pixel in the bottom line. You can also change the transparent color by clicking directly on a pixel in the selected image.

When an image has a transparent color, any pixels in the image of that color are not rendered in that color, but instead appear transparent, allowing whatever is behind the image to show through.

If the image is an icon, Transparent color appears grayed and the transparent color is set to clNone. This is because icons are already masked.

Fill color

Use the Fill color drop-down to specify a color that is added around the edges of the selected image when it is smaller than the dimensions indicated by the Height and Width properties of the image list control.

This control is grayed if the selected image completely fills the dimensions specified by the image list (that is, if it is at least as big as the Height and Width properties). This control is also grayed for icon images, because icons act like masks with any outer boundaries transparent.

Options

Use the Options radio buttons to indicate how the image list should render the selected image if it does not fit exactly in the dimensions specified by the image list's Height and Width properties. (These buttons are disabled for icons.)

SettingDescription

Crop Displays the section of the image beginning at the top-left, extending the image list width and height towards the bottom-right.

Stretch Causes the entire image to stretch so that it fits the image list width and height.

Center Centers the image within the image list width and height. If the image width or height is larger than the image list width or height, the image may be clipped.

Images

Contains a preview list view of all the images in the image list, and controls for adding or deleting images from the list. Each image is displayed within a 24x24 area for easier viewing of multiple images. Beneath each image is a caption that indicates the zero-based position of the image within the image list. You can edit the caption to change an image's position in the list or drag the image to its new position.

Add

Displays the Add Images dialog box, which lets you select one or more bitmaps or icons to add to the image list. The images then appear highlighted in the preview list view and their captions are assigned sequential values in the image list.

If a bitmap is larger than the image list width or height by even increments, a prompt appears asking whether the ImageList editor should divide the bitmap into several images. This is useful for toolbar bitmaps, which are usually composed of several small images in a sequence and stored as one larger bitmap.

Delete

Removes the selected images from the image list. All images left after clicking Delete are repositioned so they are a contiguous zero-based list.

Clear

Removes all images from the image list.

Export

Allows you to save the selected image to a file. This file contains the bitmap in its currently altered state, including any cropping or stretching.

3.2.12.17 ListView Items Editor

Use the ListView Items editor at design time to add or delete the items displayed in a listview component. You can add or delete new items and sub-items, and you can set the caption and image index for each item in the ListView Items editor.

To display the ListView Items editor, select the TListView object and double-click the Items property value in the Object Inspector.

Using the ListView Items editor

The ListView Items editor contains an Items group box with an Items list box, a New Item button, a New SubItem button, and a Delete button. When you first add a listview control to a form, the Items list box is empty and the New SubItem and Delete buttons are disabled. When you enter or change item properties for a selected item, the Apply button is enabled so that you can activate changes immediately.

The ListView Items editor also contains an Item Properties group box for setting the properties of the listview item currently selected in the Items list box. The Item Properties group box contains a Caption edit box and an Image Index edit box.

Items group box

Create and delete listview items and subitems in the Items group box. To create a new item, click New Item. A default item caption appears in the Items list box. Specify an item's properties, including its caption, in the Items Properties group box. When you create a new item, or select an existing item, the New SubItem button is enabled so that you can nest items within other items in the listview. If the Items list box contains items, the Delete button is also enabled. To delete an item, select it in the Items list box and click Delete.

Item Properties group box

Set the properties for a selected item in the Item Properties group box. Enter a name for the item in the Caption edit box. As you enter the name, it changes in the Items list box.

To display an image to the left of an item that is not currently selected, specify the index number of the image in the Image Index edit box. To suppress image display, set Image Index to -1 (the default).

3.2.12.18 New Standard Action Classes dialog box

Use the New Standard Action Classes dialog box to add a predefined action to your action list. Standard actions perform common tasks such as navigating datasets, managing the windows in an MDI application, or working with the Windows clipboard. Each standard action performs a specific function when invoked, and enables or disables any linked controls as appropriate.

Choose the action you want to add from the list and click OK. For a description of each predefined action class, see Predefined Action Classes.

3.2.12.19 String List editor

Use the String List editor at design time to add, edit, load, and save strings into any property that has been declared as TStrings.

To open the String List editor:

1. Place a component that uses a string list on the form.
2. With that component selected, do one of the following:

- Click the ellipsis in the Value column for any property that has been declared as TStrings, such as the Items property of the ComboBox property.
- Double-click the word (TStrings) in the Value column for any property that has been declared as TStrings.

Note: If the property is a value list, the Value List Editor

is displayed.

Code editor button

To convert the list to text, click Code Editor. The list is displayed on a separate page in the editor where you can edit is using all of the editing commands.

String list editor context menu

The String List editor context menu (right-click on the editor) contains the following commands:

Load

Click Load to display the **Load String List** dialog box, where you can select an existing file to read into the String List editor.

Save

Click Save to write the current string list to a file. The product opens the **Save String List** dialog box, where you can specify a directory and file name.

See Also

Value List Editor (see page 928)

Load String List (see page 932)

Save String List (see page 936)

3.2.12.20 TreeView Items Editor

Use the TreeView Items editor at design time to add items to a tree view component, delete items from a tree view component, or load images from disk into a tree view component. You can specify the text associated with individual tree view items, and set the image index, selected index, and state index for items.

To display the TreeView Items editor, select the TTreeView object and double-click the Items property value in the Object Inspector.

Using the TreeView Items editor

The TreeView Items editor contains an Items group box with an Items list box, a New Item button, a New SubItem button, a Delete button, and a Load button. When you first add a tree view control to a form, the Items list box is empty, and the New SubItem and Delete buttons are disabled. When you enter or change item properties for a selected item the Apply button is enabled so that you can activate changes immediately.

The TreeView Items editor contains a SubItems group box with a SubItems list box, an Add SubItem button and a Delete button.

The TreeView Items editor also contains an Item Properties group box for setting the properties of the tree view item currently selected in the Items list box. The Item Properties group box contains a Text edit box, and Image Index edit box, a Selected Index edit box, and a State Index edit box.

Items group box

Create, load, and delete tree view items and subitems in the Items group box. To load a set of existing tree view items from disk, click Load. To create a new item, click New Item. Default text for the item appears in the Items list box. Specify an item's properties, including its text, in the Items Properties group box.

When you create a new item, or select an existing item, the New SubItem button is enabled so that you can nest items within other items in the tree view. If the Items list box contains items, the Delete button is also enabled. To delete an item, select it in the Items list box and click Delete.

Item Properties group box

Set the properties for a selected item in the Item Properties group box. Enter text for the item in the Text edit box. As you enter the name, it changes in the Items list box.

To display an image to the left of an item that is not currently selected, specify the index number of the image in the Image Index edit box. To suppress image display, set Image Index to -1 (the default).

To display an image to left of a selected item, specify the index number of the image in the Selected Index edit box. The index is zero-based. To suppress image display, set Selected Index to -1 (the default).

To display an additional image to the left of an item, specify the index number of the image in the State Index edit box. The index number represents an index into the StateImages property of the listview component. The index is zero-based. To suppress image display, set State Index to -1 (the default).

3.2.12.21 Value List editor

Use the Value List editor at design time to add, edit, load, and save name-value pairs into any property that has been declared as TStrings.

To open the Value List editor:

1. Place a component that uses a string list on the form.
2. With that component selected, do one of the following:

- Click the ellipsis in the Value column for any property that has been declared as TStrings.
- Double-click the word (TStrings) in the Value column for any property that has been declared as TStrings.

To add items to the value list, type the name of the item in the Key column and its value in the value column. When you click OK, the string list is saved.

To convert the list to text, click Code Editor. The list is displayed on a separate page in the editor where you can edit it using all of the editing commands.

Value List editor context menu

The Value List editor context menu (right-click on the editor) contains the following commands:

Load

Click Load to display the **Load String List** dialog box, where you can select an existing file to read into the Value List editor.

Save

Click Save to write the current string list to a file. The product opens the **Save String List** dialog box, where you can specify a directory and file name.

See Also

Load String List (↗ see page 932)

Save String List (↗ see page 936)

3.2.12.22 Input Mask editor

Use the Input Mask editor to define an edit box that limits the user to a specific format and accepts only valid characters. For example, in a data entry field for telephone numbers you might define an edit box that accepts only numeric input. If a user then tries to enter a letter in this edit box, your application will not accept it.

Use the Input Mask editor to edit the EditMask property of the MaskEdit component.

Opening the Input Mask editor

To open the Input Mask editor:

1. Place a MaskEdit component on the form.
2. With that component selected, do one of the following:

- Click the ellipsis button in the Value column for the EditMask property.
- Double-click the Value column for the EditMask property.

Input mask

Define your own masks for the edit box. You can use special character to specify the mask; for a listing of those characters, see the EditMask property.

The mask consists of three fields separated by semicolons. The three fields are:

- The mask itself; you can use predefined masks or create your own.
- The character that determines whether or not the literal characters of the mask are saved as part of the data.
- The character used to represent a blank in the mask.

Character for Blanks

Specify a character to use as a blank in the mask. Blanks in a mask are areas that require user input.

This edit box changes the third field of your edit mask.

Save Literal Characters

Check to store the literal characters from the edit mask as part of the data. This option affects only the Text property of the MaskEdit component. If you save data using the EditText property, literal characters are always saved.

This check box toggles the second field in your edit mask.

Test Input

Use Test Input to verify your mask. This edit box displays the edit mask as it will appear on the form.

Sample Masks

Select a predefined mask to use in the MaskEdit component. When you select a mask from this list, the product places the predefined mask in the Input Mask edit box and displays a sample in the Test Input edit box. To display masks appropriate to your country, choose the Masks button.

Masks button

Choose Masks to display the Open Mask File dialog box, where you choose a file containing the sample masks shown in the Sample Masks list box.

3.2.12.23 Insert Object dialog box

Use the Insert Object dialog box at design time to insert an OLE object into an OleContainer component. The OleContainer component enables you to create applications that can share data with an OLE server application. After you insert an OLE server object in your application, you can double-click the OleContainer component to start the server application.

Select whether or not you want to create a new file using the associated OLE server or use an existing file. If you use an existing file, it must be associated with an application that can act as an OLE server.

Create New

Choose Create New to specify that you want to launch a server application to create a new OLE object. After choosing Create New, the ObjectType list box is displayed.

Create From File

Choose Create From File to specify that the OLE object has already been saved as a file. After choosing Create From File, the File, Browse and Link controls are displayed.

Object Type

Select an application that you want to use as the OLE server. This list box displays all available applications that can be used as an OLE server. After you select an application, the product launches that application.

File

Enter the fully qualified path for the file you want to insert into your application. The file you choose must be associated with an application that can be used as an OLE server.

Note: This option is available only when you have selected the Create From File radio button.

Browse

Click Browse to display the **Browse** dialog box, where you can select a file to use as the OLE server.

Note: This option is available only when you have selected the Create From File radio button.

Link

Check Link to link the object on the form to a file. When an object is linked, it is automatically updated whenever the source file is modified. When Link is unchecked, you are embedding the object, and changes made to the original are not reflected in your container.

Display As Icon

Check to display the inserted object as an icon on the form. When this option is checked, the Change Icon button is displayed.

Change Icon

Click Change Icon to open the Change Icon dialog box, where you can specify an icon and label for the object you inserted onto the form.

Note: This option is available only when you have selected the Create From File radio button.

See Also

Browse (see page 915)

3.2.12.24 Loading an image at design time

Use the **Picture Editor** to load images onto any of several graphic-compatible components and to specify an image to represent a form when it is minimized at runtime.

Each graphic-compatible component has a property that uses the **Picture Editor**.

To load an image at design time:

1. Add a graphic-compatible component (such as TImage) to your form.
2. To automatically resize the component so that it fits the graphic, set the component's **AutoSize** property to true before you load the graphic.
3. In the **Object Inspector**, select the property that uses the **Picture Editor**.
4. Either double-click in the Value column, or choose the ellipsis button to open the Picture editor.

(To open the **Picture Editor** from an Image component, you can also double-click the component in the form.)

1. Choose the Load button to open the **Load Picture** dialog box.
2. Use the **Load Picture** dialog box to select the image you want to display, then choose OK.

The image you choose is displayed in the **Picture Editor**.

1. Choose **OK** to accept the image you have selected and exit the **Picture Editor** dialog box.
2. The image appears in the component on the form.

   Note: When loading a graphic into an Image component, you can automatically resize the graphic so that it fits the component by setting the Image component's Stretch property to true. (Stretch has no effect on the size of icon (.ICO) files.)

See Also

Picture Editor (see page 934)

Load Picture (see page 931)

3.2.12.25 Load Picture dialog box

Use the Load Picture dialog box to select an image to add to any of the graphic-compatible components and to specify an icon for your form.

To open the Load Picture dialog box, in the Picture editor, click Load.

Dialog box options

The following options are available for the Load Picture dialog box:

File name

Enter the name of the file you want to load or wildcards to use as filters in the Files list box.

Files (main list box)

Displays the files in the current directory that match the wildcards in the File Name edit box or the file type in the Files of Type combo box.

Files of Type

Choose a filter to display the different types of image files. By default, the icon files (*.ICO) for the current directory are displayed in the Files list box.

Directories

Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File name edit box or the file type in the Files of Type combo box appear in the Files list box.

Drives

Select the current drive. The directory structure for the current drive appears in the Directories list box.

3.2.12.26 Load String List dialog box

Use the **Load String List** dialog box to select a text file to load into a property of type TStrings.

To open this dialog box:

1. Bring up the **String List Editor**.
2. Right-click and choose Load.

Load String List options

The following options are available in the **Load String List** dialog box:

File name

Enter the name of the file you want to load or wildcards to use as filters in the Files list box.

Files (main list box)

Displays the files in the current directory that match the wildcards in the File Name edit box or the file type in the Files Of Type combo box.

Files Of Type

Choose a filter to display the different types of files. By default, the text files (*.txt) for the current directory are displayed in the Files list box.

Directories

Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File name edit box or the file type in the Files of Type combo box appear in the Files list box.

Drives

Select the current drive. The directory structure for the current drive appears in the Directories list box.

See Also

String List Editor (↗ see page 927)

3.2.12.27 Masked Text editor

Use the Mask Test editor to enter Values into the edit mask.

Use the Masked Text editor to edit the Text property of the MaskEdit component.

To open the Masked Text editor:

1. Place an MaskEdit component on the form.
2. With that component selected, do one of the following:

- Click the ellipsis button in the Value column for the Text property.
- Double-click the Value column for the Text property.

Input Text edit box

Enter initial values for the MaskEdit component. You can overwrite these values at runtime.

Edit Mask label

Displays the mask definition for the current component.

3.2.12.28 Notebook editor

Use the Notebook editor to add, edit, remove, or rearrange pages in either a TabbedNotebook component or Notebook component. You can also use the Notebook editor to add or edit Help context numbers for each notebook page.

The Notebook editor displays the current pages of the notebook in their current order, and it also displays the Help context associated with that page.

To open the Notebook editor:

1. Place a Notebook component or TabbedNotebook component from the Win 3.1 Component palette page on the form.
2. With that component selected, do one of the following:

- Click the ellipsis button in the Value column for the Pages property.
- Double-click the Value column for the Pages property.

Edit

Click **Edit** to open the **Edit Page** dialog box, where you can modify the page name and Help context number for the selected notebook page.

Add

Click **Add** to open the **Add Page** dialog box, where you can create a new notebook page.

Delete

Click **Delete** to remove the selected page from the notebook.

Move Up/Move Down

Click Move Up or Move Down to rearrange the order of the selected page or pages.

See Also

Edit Page (↗ see page 923)

Add Page (↗ see page 922)

3.2.12.29 Open dialog box

Use the Open dialog box at design time to load a multimedia file into the MediaPlayer component.

To open the Open dialog box:

1. Place a MediaPlayer component on the form.
2. With that component selected, do one of the following:

- Click the ellipsis button in the Value column for any of the properties listed below.
- Double-click the Value column for either of the properties listed below.

Open dialog box options

The following options are available for the Open dialog box":

File name

Enter the name of the file you want to load or wildcards to use as filters in the Files list box.

Files

Displays the files in the current directory that match the wildcards in the File Name edit box or the file type in the Files Of Type combo box.

Files Of Type

Choose the type of file you want to load. By default, all files in the current directory are displayed. However, you can limit the display to wave files, midi files, or Windows video files.

Directories

Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File Name edit box or the file type in the Files Of Type combo box appear in the Files list box.

Drives

Select the current drive. The directory structure for the current drive appears in the Directories list box.

3.2.12.30 Paste Special dialog box

Use the Paste Special dialog box to insert an object from the Windows clipboard into your OLE container.

Source label

Displays the path of the file you are going to paste.

Paste/Paste Link Radio

Select Paste to embed the object on the form. When you embed an object on a form, your container application stores all the information for the object. It is not necessary to have an external file.

Select Paste Link to link the object to the form. When you link an object to a form, the main source is stored in a file so that when you update the object, the source file is also updated.

As

Lists the type of application object you are pasting. The application listed is the source application from which you received the object that you are pasting.

3.2.12.31 Picture editor

Use the Picture editor to select an image to add to any of the graphic-compatible components and to specify an icon for your form.

To open the Picture editor:

1. Place a graphic-compatible component (such as TImage) on the form.
2. With that component selected, do one of the following:
3. Click the ellipsis button in the Value column for properties (such as the Picture property of TImage) related to editing the picture.
4. Double-click the Value column for properties related to editing the picture.

   Note: To open the Picture editor from an Image component, you can also double-click the component on the form.

Picture editor commands

The Picture editor provides the following commands:

Load

Display the **Load Picture** dialog box, where you can select an existing file to read into the Picture editor. For more information about loading images into the Picture editor, see the information on loading an image at design time.

Save

Display the **Save Picture As** dialog box, where you can specify a directory and file name in which to store the image.

Clear

Remove the association between the current image and the selected component.

See Also

Load Picture (↗ see page 931)

Loading an image (↗ see page 931)

Save Picture As (↗ see page 935)

3.2.12.32 Save Picture As dialog box

Use the **Save Picture As** dialog box to store the image loaded in the **Picture Editor** into a new file or directory.

To open the **Save Picture As** dialog box, in the **Picture Editor**, click Save As.

Save Picture As options

The following options are available in the **Save Picture As** dialog box:

File name

Enter the name of the file you want to load or wildcards to use as filters in the Files list box.

Files (main list box)

Displays the files in the current directory that match the wildcards in the File Name edit box or the file type in the Files of Type combo box.

Files of Type

Choose filter to display the different types of image files. By default, the icon files (*.ICO) for the current directory are displayed in the Files list box.

Directories

Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File name edit box or the file type in the Files of Type combo box appear in the Files list box.

Drives

Select the current drive. The directory structure for the current drive appears in the Directories list box.

See Also

Picture Editor (↗ see page 934)

3.2.12.33 Save String List dialog box

Use the Save string list dialog box to store the string list from the String List editor into a text file.

To open this dialog box:

1. Bring up the **String List Editor**.
2. Right-click and choose Save.

Save String List options

The following options are available in the Save String List dialog box:

File name

Enter the name of the file you want to save or wildcards to use as filters in the Files list box.

Files (main list box)

Displays the files in the current directory that match the wildcards in the File Name edit box or the file type in the Files of Type combo box.

Files of Type

Choose a filter to display the different types of files. By default, the text files (*.txt) in the current directory are displayed in the Files list box.

Directories

Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File name edit box or the file type in the Files of Type combo box appear in the Files list box.

Drives

Select the current drive. The directory structure for the current drive appears in the Directories list box.

See Also

String List Editor (↗ see page 927)

3.2.13 Run

Topics

Name	Description
Add Address Breakpoint or Add Data Breakpoint (↗ see page 938)	**Run ▶ Add Breakpoint ▶ Address Breakpoint** **Run ▶ Add Breakpoint ▶ Data Breakpoint** Sets a breakpoint on either an address or a data item, and to change the properties of an existing breakpoint. The title might also be **Address Breakpoint Properties** or **Data Breakpoint Properties**, if you accessed the dialog box through the context menu Properties command.

Add Source Breakpoint (↗ see page 940)	**Run ▶ Add Breakpoint ▶ Source Breakpoint** Sets a breakpoint on a line in your source code or to change the properties of an existing breakpoint. The dialog box title can also be **Source Breakpoint Properties** or **Address Breakpoint Properties**, depending on how it is accessed.
Attach to Process (↗ see page 941)	**Run ▶ Attach to Process** Debugs a process that is currently running on the local or remote computer.
Change (↗ see page 941)	Assigns a new value to the data item currently selected on the **Data** tab in the Debug Inspector.
Debug Inspector (↗ see page 942)	**Run ▶ Inspect** Inspects the following types of data: arrays, classes, constants, functions, pointers, scalar variables, and interfaces. The **Debug Inspector** contains three areas:
Debugger Exception Notification (↗ see page 943)	This dialog box appears when the program you are debugging raises a language exception or operating system exception, and you have set options that instruct the debugger to handle exceptions on the **Language Exceptions** and **Native OS Exceptions** pages of **Tools ▶ Options ▶ Debugger Options**. The message format is: `Project <project-name> raised exception class <yyyy> with message <message-text>.` If `yyyy` is a class name, the exception is a language exception. If `yyyy` is a hexadecimal value, the exception is an operating system exception.
Evaluate/Modify (↗ see page 944)	**Run ▶ Evaluate/Modify** Evaluates or changes the value of an existing expression or property. This is useful if you want to test a code correction without having to exit the debugger, change the source code, and recompile the program.
Find Package Import (↗ see page 944)	**Run ▶ Run (F9)** The **Find Package Import** dialog appears when your application cannot locate one of the runtime packages specified in **Project ▶ Options ▶ Packages** dialog.
Inspect (↗ see page 945)	**Run ▶ Inspect** Used to enter the expression that you want to inspect in the **Debug Inspector**.
Load Process Environment Block (↗ see page 945)	**Run ▶ Load Process ▶ Environment Block** Indicates which environment variables are passed to your application while you are debugging it.
Load Process Local (↗ see page 946)	**Run ▶ Load Process ▶ Local** Passes command-line parameters to your application, specify a host executable for testing a DLL, or load an executable into the debugger.
Load Process Remote (↗ see page 946)	**Run ▶ Load Process ▶ Remote** Connects to a remote computer running the remote debug server and start a remote debugging session.
Load Process Symbol Tables (↗ see page 947)	**Run ▶ Load Process ▶ Symbol Tables** Specifies the location of the symbols tables to be used during debugging.
New Expression (↗ see page 948)	Inspects a new expression. Enter the expression or select a previously entered expression from the drop-down list.
Debug session in progress. Terminate? (↗ see page 948)	Your application is running during a debugging session and will be terminated if you click **OK**. When possible, click **Cancel** and terminate your application normally.
Type Cast (↗ see page 948)	Specifies a different data type for an item you want to inspect. Type casting is useful if the Debug Inspector contains a symbol for which there is no type information, and when you want to explicitly set the type for untyped pointers.
Watch Properties (↗ see page 948)	**Run ▶ Add Watch** Adds a watch or to change the properties of an existing watch. The watch appears in the Watch List. **Note:** The format specifiers listed in the Watch Properties dialog box will depend on the format specifiers supported by the current evaluator. Not all of the format specifiers listed below will be available for every evaluator. In most cases the evaluator is specified by the Personality.
Detach From Program (↗ see page 949)	**Run ▶ Detach From Program** Disconnects the debugger from the current (running) program and refocuses on the IDE.
Load Process (↗ see page 949)	**Run ▶ Load Process** Opens the **Load Process** dialog box. This command provides a separate UI for loading an arbitrary process into the debugger. **Tip:** Press F1 in any item in the list to bring up a help page on the selected topic.

Parameters (↗ see page 950)	**Run ▶ Parameters** Specifies the command-line parameters to pass to your application. Opens **Project ▶ Options ▶ Debugger** dialog.
Program Reset (↗ see page 950)	**Run ▶ Program Reset** Terminates the application or process that is currently under the debugger's control.
Register ActiveX Server (↗ see page 950)	**Run ▶ Register ActiveX Server** Registers an "in-process" automation server into the registry. An In-Process Automation Server is a set of COM Automation Objects for connecting to databases, executing SQL statements and PL/SQL blocks, and accessing the results. In-process automation objects are inside of .dlls; out-of-process automation servers are applications (.exes). ActiveX is used when you create a new application and then select **File ▶ New ▶ Other ▶ Active X ▶ Automation Server Object**.
Run (↗ see page 950)	**Run ▶ Run** Compiles any changed source code and, if the compile is successful, executes your application, allowing you to use and test it in the IDE .
Run To Cursor and Run Until Return (↗ see page 950)	**Run ▶ Run To Cursor** **Run ▶ Run Until Return** Executes the current program you are working on and stops either at the cursor location or when the function returns.
Show Execution Point (↗ see page 951)	**Run ▶ Show Execution Point** Opens the **Code Editor** window and bring the line of code that contains the current execution point to the front. Use this command if you closed or minimized the **Code Editor** window because the debugger automatically displays the execution point in the **Code Editor**.
Step Over (↗ see page 951)	**Run ▶ Step Over** Tells the debugger to execute the next line of code. If the line contains a function, Step Over executes the function and then stops at the first line after the function.
Trace Into (↗ see page 951)	**Run ▶ Trace Into** Tells the debugger to execute the next line of code. If the line contains a function, Trace Intro executes the function and then stops at the first line of code inside the function.
Trace to Next Source Line (↗ see page 951)	**Run ▶ Trace to Next Source Line** Executes a single source line.
Unregister ActiveX Server (↗ see page 951)	**Run ▶ Unregister ActiveX Server** Unregisters an "in-process" automation server from the registry.

3.2.13.1 Add Address Breakpoint or Add Data Breakpoint

Run ▶ Add Breakpoint ▶ Address Breakpoint

Run ▶ Add Breakpoint ▶ Data Breakpoint

Sets a breakpoint on either an address or a data item, and to change the properties of an existing breakpoint. The title might also be **Address Breakpoint Properties** or **Data Breakpoint Properties**, if you accessed the dialog box through the context menu Properties command.

Item	Description
Address	Specifies the address for the address breakpoint. When the address is executed, the program execution will halt if the condition (optional) evaluates to true and the pass count (optional) has been completed. If the address can be correlated to a source line number, the address breakpoint is created as a source breakpoint.
Length (for data breakpoint only)	Specifies the length of the data breakpoint, beginning at “Address.” The length is automatically calculated for standard data types.

Condition	Specifies a conditional expression that is evaluated each time the breakpoint is encountered. Program execution stops when the expression evaluates to true. Enter a conditional expression to stop program execution. Enter any valid language expression. All symbols in the expression must be accessible from the breakpoint's location. Functions are valid if they return a Boolean type. For data breakpoints, if no condition is set, the breakpoint is hit when any change is made to the data in the range specified in the Length field.
Pass count	Stops program execution at a certain line number after a specified number of passes. Enter the number of passes. The debugger increments the pass count each time the line containing the breakpoint is encountered. When the pass count equals the specified number, the debugger pauses program execution. For example, if the pass count is set to 3, you will see 0 of 3, 1 of 3, 2 of 3, then 3 of 3 in the pass count. Program execution stops at 3 of 3. Because the debugger increments the count with each pass, you can use the count to determine the iteration of a loop that fails. Set the pass count to the maximum loop count and run your program. When the program fails, you can calculate the number of loop iterations by examining the number of passes that occurred. When you use pass counts with conditions, program execution pauses the *n*th time that the conditional expression is true. the debugger decrements the pass count only when the conditional expression is true.
Group	Creates a breakpoint group, and makes this breakpoint a member of the group. Using breakpoint groups is useful for performing a similar set of actions on all breakpoints within a group. To create a group, enter a name in this field. To use an existing group, select a group from the drop-down list.
Advanced	Expands the dialog box to include fields for associating actions with breakpoints.
Break	Halts execution; the traditional and default action of a breakpoint.
Ignore subsequent exceptions	Ignores all subsequent exceptions raised by the current process during the current debug session (the debugger will not stop on any exception). Use this with **Handle subsequent exceptions** as a pair. You can surround specific blocks of code with the *Ignore/Handle* pair to skip any exceptions which occur in that block of code.
Handle subsequent exceptions	Handles all subsequent exceptions raised by the current process during the current debug session (the debugger will stop on exceptions based on the current exception settings in **Tools▶Options▶Debugger Options▶Language Exceptions**. This option does stop on all exceptions. Use it to turn on normal exception behavior after another breakpoint disabled normal behavior using the **Ignore subsequent exceptions** option.
Log message	Writes the specified message to the event log. Enter the message that you want to log.
Eval expression	Evaluates the specified expression and, because **Log result** is checked by default, writes the result of the evaluation to the event log. Uncheck **Log result** to evaluate without logging.
Log result	If text is entered in **Eval expression**, writes the result of the evaluation in the to the event log. If unchecked, the evaluation is not logged.
Enable group	Enables all breakpoints in the specified group. Select the group name.
Disable group	Disables all breakpoints in the specified group. Select the group name.
Log Call Stack	Displays all or part of the call stack in the **Event Log** window when a breakpoint is encountered. **Entire Stack** displays all of the call stack. **Partial Stack** displays only the number of frames specified in **Number of frames**.

See Also

Setting and Modifying Breakpoints (see page 118)

Add Module Load Breakpoint dialog box (see page 1019)

Add Source Breakpoint dialog box (see page 940)

3.2.13.2 Add Source Breakpoint

Run ▶ Add Breakpoint ▶ Source Breakpoint

Sets a breakpoint on a line in your source code or to change the properties of an existing breakpoint. The dialog box title can also be **Source Breakpoint Properties** or **Address Breakpoint Properties**, depending on how it is accessed.

Item	Description
Filename	Specifies the source file for the source breakpoint. Enter the name of the source file for the breakpoint.
Line number	Sets or changes the line number for the breakpoint. Enter or change the line number for the breakpoint.
Condition	Specifies a conditional expression that is evaluated each time the breakpoint is encountered. Program execution stops when the expression evaluates to true. Enter a conditional expression to stop program execution. Enter any valid language expression. All symbols in the expression must be accessible from the breakpoint's location. Functions are valid if they return a Boolean type.
Pass count	Stops program execution at a certain line number after a specified number of passes. Enter the number of passes. The debugger increments the pass count each time the line containing the breakpoint is encountered. When the pass count equals the specified number, the debugger pauses program execution. For example, if the pass count is set to 3, you will see 0 of 3, 1 of 3, 2 of 3, then 3 of 3 in the pass count. Program execution stops at 3 of 3. Because the debugger increments the count with each pass, you can use them to determine which iteration of a loop fails. Set the pass count to the maximum loop count and run your program. When the program fails, you can calculate the number of loop iterations by examining the number of passes that occurred. When you use pass counts with conditions, program execution pauses the *n*th time that the conditional expression is true. the debugger decrements the pass count only when the conditional expression is true.
Group	Creates a breakpoint group, of which this breakpoint becomes a member. To use an existing group, select a group from the drop-down list. Using breakpoint groups is useful for performing a similar set of actions on all breakpoints within a group.
Keep existing Breakpoint	When checked, the breakpoint will not be modified, and a new breakpoint will be created with the changes made to the existing breakpoint.
Advanced	Expands the dialog box to include fields for associating actions with breakpoints.
Break	Halts execution; the traditional and default action of a breakpoint.
Ignore subsequent exceptions	Ignores all subsequent exceptions raised by the current process during the current debug session (the debugger will not stop on any exception). Use this with **Handle subsequent exceptions** as a pair. You can surround specific blocks of code with the *Ignore/Handle* pair to skip any exceptions which occur in that block of code.
Handle subsequent exceptions	Handles all subsequent exceptions raised by the current process during the current debug session (the debugger will stop on exceptions based on the current exception settings in **Tools ▶ Options ▶ Debugger Options ▶ Language Exceptions**. This option does stop on all exceptions. Use it to turn on normal exception behavior after another breakpoint disabled normal behavior using the **Ignore subsequent exceptions** option.
Log message	Writes the specified message to the event log. Enter the message that you want to log.
Eval expression	Evaluates the specified expression and, because **Log result** is checked by default, writes the result of the evaluation to the event log. Uncheck **Log result** to evaluate without logging.

Log result	If text is entered in **Eval expression**, writes the result of the evaluation in the to the event log. If unchecked, the evaluation is not logged.
Enable group	Enables all breakpoints in the specified group. Select the group name.
Disable group	Disables all breakpoints in the specified group. Select the group name.
Log Call Stack	Displays all or part of the call stack in the **Event Log** window when a breakpoint is encountered. **Entire Stack** displays all of the call stack. **Partial Stack** displays only the number of frames specified in **Number of frames**.

See Also

Setting and Modifying Breakpoints (see page 118)

Add Address Breakpoint or Add Data Breakpoint dialog box (see page 938)

Add Module Load Breakpoint dialog box (see page 1019)

3.2.13.3 Attach to Process

Run ▶ Attach to Process

Debugs a process that is currently running on the local or remote computer.

Item	Description
Debugger	Select the appropriate debugger from the list of registered debuggers. If you choose the CodeGear .NET debugger, only managed processes are displayed **Running Processes** list. If you choose the CodeGear Win32 debugger, both managed and unmanaged processes are displayed.
Remote Machine	The name of the remote machine that is running the application that you want to debug.
Running Processes	Lists the processes currently running on the local or if specified, the remote computer. Note that the Remote Server must be running.
PID	Lists the process identifier of the process.
Path	Lists the location of the process.
Show System Processes	Displayed if the **Debugger** option indicates the CodeGear Win32 debugger. Includes system processes in the **Running Processes** list.
Pause After Attach	Pauses the process after the debugger attaches to it and displays the **CPU View**. You will need to run, step, or trace to resume execution.
Refresh	Refreshes and redisplays the list of running processes.
Attach	Attaches the debugger to the selected process and, if **Pause After Attach** is enabled, displays the **CPU** window. The **Attach** button is disabled for the IDE itself or for any process that you have already attached to with the debugger.

3.2.13.4 Change

Assigns a new value to the data item currently selected on the **Data** tab in the Debug Inspector.

3.2.13.5 Debug Inspector

Run ▶ Inspect

Inspects the following types of data: arrays, classes, constants, functions, pointers, scalar variables, and interfaces.

The **Debug Inspector** contains three areas:

Area	Description
Top pane	Displays the name, type, and address or memory location of the inspected element, if available. When inspecting a function call that returns an object, record, set, or array, the debugger displays “In debugger” in place of the temporarily allocated address.
Middle pane	Displays one or more of the following tabs, depending on the type of data you inspect: **Data** - Shows data names (or class data members) and current values. **Methods** - Displayed only when you inspect a class, or interface and shows the class methods (member functions) and current address locations. **Properties** - Displayed only when you inspect an object class with properties and shows the property names and current values.
Bottom pane	Displays the data type of the item currently selected in the middle pane.
Status bar	Displays the data type of the element being inspected.

Context Menu

Right-click the **Debug Inspector** to display the following commands.

Item	Description
Change	Lets you assign a new value to a data item. An ellipsis (…) appears next to an item that can be changed. You can click the ellipsis as an alternative to choosing the change command. This command is only enabled when you can modify the data item being inspected.
Show Inherited	Switches the view in the **Data**, **Methods**, and **Properties** panes between two modes: one that shows all intrinsic and inherited data members or properties of a class, or one that shows only those declared in the class.
Show Fully Qualified Names	Shows inherited members using their fully qualified names.
Sort By	Sorts the data elements displayed in the **Debug Inspector** by their name or by the order in which they were declared in the code.
Bind to Object	(For Delphi.NET and managed code only) Attaches the Inspector to the data item currently displayed. When the **Debug Inspector** is bound to an object, “(Bound)” is appended to the name of the inspected element displayed at the top of the dialog box. Once an Inspector is bound, the displayed value cannot be subinspected. The **Debug Inspector** remains bound until you close it or until you open another **Debug Inspector** by selecting the Inspect, Descend, New Expression, or Typecast command from the context menu.
Inspect	Opens a new **Debug Inspector** on the data element you have selected. This is useful for seeing the details of data structures, classes, and arrays.
Descend	Same as the Inspect command, except the current Debug Inspector is replaced with the details that you are inspecting (a new **Debug Inspector** is not opened). To return to a higher level, use the history list.
New Expression	Lets you inspect a new expression.

Type Cast	Lets you specify a different data type for an item you want to inspect. Type casting is useful if the **Debug Inspector** contains a symbol for which there is no type information, and when you want to explicitly set the type for untyped pointers.
Dockable	Toggles whether the **Debug Inspector** window is dockable.
Stay On Top	Keeps the window visible when out of focus.

See Also

Inspecting and Changing the Value of Data Elements (see page 122)

3.2.13.6 Debugger Exception Notification

This dialog box appears when the program you are debugging raises a language exception or operating system exception, and you have set options that instruct the debugger to handle exceptions on the **Language Exceptions** and **Native OS Exceptions** pages of **Tools**▶**Options**▶**Debugger Options**.

The message format is:

```
Project <project-name> raised exception class <yyyy> with message <message-text>.
```

If `yyyy` is a class name, the exception is a language exception. If `yyyy` is a hexadecimal value, the exception is an operating system exception.

Item	**Description**
Ignore this exception type	Causes the debugger to ignore this type of language exception or OS exception and sets the corresponding check box in the **Exception Types to Ignore** list on the **Tools**▶**Options**▶**Debugger Options**▶**Language Exceptions** page. If you choose to ignore the OS exception, the **Handled by** option is set to **Debugger** on the **Native OS Exceptions** page for all exception ranges that include the exception that was raised.
Inspect exception object	Displayed only on exceptions when using the CodeGear .NET Debugger. Displays the **Debug Inspector** dialog box for the exception object if you click **Break** to halt execution. This option has no effect if you click **Continue**.
Show CPU view	This option is displayed only if the location of the exception does not correspond to a source location. Displays the **CPU View** if you click **Break** to halt execution. If **Show CPU view** is displayed and you do not check it, the IDE traverses the call stack looking for a call in the stack that contains source and will show you the first call found that has source. This option has no effect if you click **Continue**.
Break	Halts program execution where the exception occurred and positions the **Code Editor** to that line of code.
Continue	Continues program execution.

Tip: To copy this or other messages to the clipboard, type CTRL+C

.

Note: In some cases, the state of the program will prevent you from continuing, and you will repeatedly see the Debugger Exception Notification

dialog box. If this occurs, click **Break** and then choose **Run**▶**Program Reset** to end the current program run and release it from memory.

3.2.13.7 Evaluate/Modify

Run ▸Evaluate/Modify

Evaluates or changes the value of an existing expression or property. This is useful if you want to test a code correction without having to exit the debugger, change the source code, and recompile the program.

Item	Description
Evaluate	Evaluates the expression in the **Expression** edit box and displays its value in the **Result** edit box.
Modify	Changes the value of the expression in the **Expression** edit box using the value in the **New Value** edit box.
Watch	Creates a watch for the expression you have selected.
Inspect	Opens a new **Debug Inspector** on the data element you have selected. This is useful for seeing the details of data structures, classes, and arrays.
Expression	Enter the variable, field, array, or object to evaluate or modify. By default, the word at the cursor position in the **Code Editor** is placed in the **Expression** edit box. You can accept this expression, enter another one, or choose an expression from the history list of previously evaluated expressions. To evaluate a function call, enter the function name, parentheses, and arguments as you would type it in your program, but leave out the statement-ending semicolon (;).
Result	Displays the value of the item specified in the **Expression** text box after you choose **Evaluate** or **Modify**.
New value	Assigns a new value to the item specified in the **Expression** edit box. Enter a new value for the item if you want to change its value.

Note: You can evaluate any valid language expression or static variables that are accessible from the current execution point.

Display Format Specifiers

By default, the debugger displays the result in the format that matches the data type of the expression. For example, Integer values are displayed in decimal format. To change the display format, type a comma (,) followed by a format specifier after the expression.

The following table describes the Evaluate/Modify format specifiers.

Specifier	Types affected	Description
,C	Char, strings	Character. Shows characters for ASCII 0 to 31 in the Delphi language #nn notation.
,S	Char, strings	String. Shows ASCII 0 to 31 in Delphi language #nn notation.
,D	Integers	Decimal. Shows integer values in decimal form, including those in data structures.
,H or ,X	Integers	Hexadecimal. Shows integer values in hexadecimal with the $ prefix, including those in data structures.
,Fn	Floating point	Floating point. Shows *n* significant digits where *n* can be from 2 to 18. For example, to display the first four digits of a floating-point value, type ,F4. If *n* is not specified, the default is 11.

3.2.13.8 Find Package Import

Run▸Run (F9)

The **Find Package Import** dialog appears when your application cannot locate one of the runtime packages specified in

Project ▶ Options ▶ Packages dialog.

Item	Description
Package Import	Lists the name of the package that cannot be located.
Browse	Displays a file browser so that you can locate the correct package name.
Remove this reference	Deletes the name of the package import from the list of runtime packages for the project.
Don't ask me this again	Specifies that RAD Studio is to proceed with loading runtime packages without displaying this dialog again.

See Also

Packages Overview (↗ see page 640)

Runtime Packages

Loading Packages in an Application

3.2.13.9 Inspect

Run ▶ Inspect

Used to enter the expression that you want to inspect in the **Debug Inspector**.

Item	Description
Expression	Enter a valid expression.

See Also

Inspecting and Changing the Value of Data Elements (↗ see page 122)

3.2.13.10 Load Process Environment Block

Run ▶ Load Process ▶ Environment Block

Indicates which environment variables are passed to your application while you are debugging it.

Item	Description
Debugger	The name of the debugger to be used. You can select CodeGear Win32 Debugger or CodeGear .NET Debugger, depending on the type of application you are debugging.
System variables	Lists all environment variables and their values defined at a system level. You cannot delete an existing system variable, but you can override it.
Add Override	Displays the **Override System Variable** dialog box, allowing you to modify an existing system variable to create a new user override. This button is dimmed until you select a variable in the **System variables** list.
User overrides	Lists all defined user overrides and their values. A user override takes precedence over an existing system variable until you delete the user override.
New	Displays the **New User Variable** dialog box allowing you to create new user override to a system variable.
Edit	Displays the **Edit User Variable** dialog box allowing you to change the user override currently selected in the **User overrides** list.

Delete	Deletes the user override currently selected in the **User overrides** list.
Include System Variables	Passes the system environment variables to the application you are debugging. If unchecked, only the user overrides are passed to your application.

3.2.13.11 Load Process Local

Run ▶ Load Process ▶ Local

Passes command-line parameters to your application, specify a host executable for testing a DLL, or load an executable into the debugger.

Item	Description
Debugger	The name of the debugger to be used. You can select either the Borland Win32 Debugger or the CodeGear .NET Debugger from the pull-down menu.
Process	Enter the path to an executable file that you want to run in the debugger. Then click **Load** to load the executable. The executable will be paused at its entry point. If there is no debug information at the entry point, the **CPU** window will be opened. Select **Run ▶ Run** to run the executable.
Parameters	Enter the command-line arguments you want to pass to your application when it starts.
Working Directory	Enter the name of the directory to use for the debugging process. By default, this is the same directory as the one containing the executable of your application.
Execute Startup Code on Load	Executes the startup code that was automatically generated when you created the project. The startup code is executed before reaching the program's main entry point.
Load	Loads the application (the process is loaded and stopped).

3.2.13.12 Load Process Remote

Run ▶ Load Process ▶ Remote

Connects to a remote computer running the remote debug server and start a remote debugging session.

Item	Description
Debugger	The name of the debugger to be used. You can select CodeGear Win32 Debugger or CodeGear .NET Debugger, depending on the type of application you are debugging.
Remote Path	Enter the path of the .exe file on the remote computer, relative to the directory that contains the remote debug server (rmtdbg105.exe).
Remote Host	Enter the name or TCP/IP address of the remote computer on which you want to run the application. The remote debug server (rmtdbg100.exe) must be running on the remote computer. If a port was specified when starting rmtdbg100.exe, enter a colon after the host name, followed by the port. For example, if you specified port 8000, specify the remote host as`somehost:8000` or `127.0.0.1:8000`. Otherwise, the default port 64447 will be used.
Parameters	Enter the command-line arguments you want to pass to your application (or the host application) when it starts.
Working Directory	Enter the name of the directory to use for the debugging process. By default, this is the same directory as the one containing the executable of your application.
Execute startup code on Load	Executes the startup code that was automatically generated when you created the project. The startup code is executed before reaching the program's main entry point.

See Also

Debugging Remote Applications (see page 125)

3.2.13.13 Load Process Symbol Tables

Run ▶ Load Process ▶ Symbol Tables

Specifies the location of the symbols tables to be used during debugging.

Item	Description
Debugger	The name of the debugger to be used. You can select CodeGear Win32 Debugger or CodeGear .NET Debugger, depending on the type of application you are debugging.
Debug symbols search path	Specifies the directory containing the symbol tables used for debugging. This path is used if you check the **Load all symbols** check box.
Load all symbols	Sets the state of the **Mappings from Module Name to Symbol Table Path** list. If checked, the list is disabled and all symbol tables are loaded by the debugger. The debugger uses the **Debug symbols search path** to search for the symbol table file associated with each module loaded by the process being debugged. If unchecked, the **Mappings from Module Name to Symbol Table Path**list is enabled and its settings are used.
Mappings from Module Name to Symbol Table Path	Displays the current mapping of each module name to a symbol table search path that is defined for the project. Use the up and down arrows (to the right of the dialog) to move the selected item up or down in the list. The debugger searches this list, in order, to find a match for the name of the module being loaded. When the debugger finds a matching module name, it uses the corresponding path to locate that module's symbol table. For example, if module **foo123.dll** is loaded, and the list shows **foo*.dll** as the first item and ***123.dll** as a later item, the debugger only uses the symbol table path for **foo*.dll**, even though both items match the module being loaded.
Load symbols for unspecified modules	Specifies whether symbol tables for modules not in the **Mappings from Module Name to Symbol Table Path** list (either explicitly or via a file mask) are loaded during debugging. If checked, the symbol tables for modules not specified will be loaded using the **Debug symbols search path**. If unchecked, symbol tables are loaded only for modules in the list.
New	Displays the **Add Symbol Table Search Path** dialog, where you can specify a module name and an associated search table path. The module and path are added to the **Mappings from Module Name to Symbol Table Path** list. Note that you can add a blank path to prevent a symbol table for a module from being loaded.
Edit	Displays the selected module and path in the **Add Symbol Table Search Path** dialog, enabling you to edit the module name or path that displays in the **Mappings from Module Name to Symbol Table Path** list.
Delete	Removes the selected module from the **Mappings from Module Name to Symbol Table Path** list.

See Also

Debugging Applications (see page 10)

Preparing Files for Remote Debugging (see page 128)

Local (see page 946)

Remote (see page 946)

3.2.13.14 New Expression

Inspects a new expression. Enter the expression or select a previously entered expression from the drop-down list.

3.2.13.15 Debug session in progress. Terminate?

Your application is running during a debugging session and will be terminated if you click **OK**. When possible, click **Cancel** and terminate your application normally.

3.2.13.16 Type Cast

Specifies a different data type for an item you want to inspect. Type casting is useful if the Debug Inspector contains a symbol for which there is no type information, and when you want to explicitly set the type for untyped pointers.

3.2.13.17 Watch Properties

Run ▶ Add Watch

Adds a watch or to change the properties of an existing watch. The watch appears in the Watch List.

Note: The format specifiers listed in the Watch Properties

dialog box will depend on the format specifiers supported by the current evaluator. Not all of the format specifiers listed below will be available for every evaluator. In most cases the evaluator is specified by the Personality.

Item	Description
Expression	Specifies the expression to watch. Enter or edit the expression you want to watch. Use the drop-down list to choose from previously entered expressions.
Group name	Specifies the group in which the selected watch resides. If you specify a new group, the new group is added and the watch is moved to the new group. Use the drop-down list to choose the name from a list of the existing watch groups.
Repeat count	Specifies the repeat count when the watch expression represents a data element, or specifies the number of elements in an array when the watch expression represents an array. When you watch an array and specify the number of elements as a repeat count, the Watch List displays the value of every element in the array.
Digits	Specifies the number of significant digits in a watch value that is a floating-point expression. Enter the number of digits. This option takes affect only when you select **Floating Point** as the Display format.
Enabled	Enables or disables the watch. Disabling a watch hides the watch from the current program run. When you disable a watch, its settings remain defined, but the debugger does not evaluate the watch. Disabling watches improves performance of the debugger because it does not monitor the watch as you step through or run your program. When you set a watch, it is enabled by default.
Allow Function Calls	Evaluates the watch even if doing so causes function calls. This option is off by default for all watches. When off, watches that would make function calls are not evaluated but instead generate the error message ”Inaccessible value.”
Character	Shows special display characters for ASCII 0 to 31 (displayed as #$0, #$1F, and so on). This format type affects characters and strings.

String	Shows characters for ASCII 0 to 31 in the Pascal #nn notation (#$0, and so on.) This format type affects characters and strings.
Decimal	Shows integer values in decimal form, including those in data structures. This format type affects integers.
Hexadecimal	Shows integer values in hexadecimal with the 0x (for C++, C#) or $ (for Delphi) prefix, including those in data structures. This format type affects integers.
Floating point	Shows integer values in floating-point notation (real numbers or numbers that can contain fractional parts).
Pointer	Used for Win32 applications only.
Record/Structure	Shows both field names and values such as (X:1;Y:10;Z:5) instead of (1,10,5).
Default	Shows the result in the display format that matches the data type of the expression. This format type affects all.
Memory Dump	Used for Win32 applications only.

Tip: By default, the debugger displays the result of a watch in the format that matches the data type of the expression. For example, integer values are displayed in decimal form. However, if you select the Hexadecimal

for an integer type expression, the display format changes from decimal to hexadecimal.

When setting up a watch on an element in a data structure (such as an array), you can display the values of consecutive data elements. For example, suppose you have an array of five integers named xarray. Type the number 5 in **Repeat Count** to see all five values of the array. To use a repeat count, however, the watch expression must represent a single data element.

To change the value of a watch expression, use the **Evaluate/Modify** dialog box.

Tip: To format a floating-point expression, select Floating Point

and enter a number for **Digits** to indicate the number of significant digits you want displayed in the Watch List.

3.2.13.18 Detach From Program

Run ▶Detach From Program

Disconnects the debugger from the current (running) program and refocuses on the IDE.

3.2.13.19 Load Process

Run ▶Load Process

Opens the **Load Process** dialog box. This command provides a separate UI for loading an arbitrary process into the debugger.

Tip: Press F1

in any item in the list to bring up a help page on the selected topic.

See Also

Load Process Environmental Block

Load Process Local (see page 946)

Load Process Remote (see page 946)

Load Process Symbol Tables (see page 947)

3.2.13.20 Parameters

Run ▶ Parameters

Specifies the command-line parameters to pass to your application. Opens **Project ▶ Options ▶ Debugger** dialog.

See Also

Parameters (↗ see page 672)

Debugger (↗ see page 836)

3.2.13.21 Program Reset

Run ▶ Program Reset

Terminates the application or process that is currently under the debugger's control.

See Also

Attach Process (↗ see page 117)

3.2.13.22 Register ActiveX Server

Run ▶ Register ActiveX Server

Registers an "in-process" automation server into the registry. An In-Process Automation Server is a set of COM Automation Objects for connecting to databases, executing SQL statements and PL/SQL blocks, and accessing the results. In-process automation objects are inside of .dlls; out-of-process automation servers are applications (.exes). ActiveX is used when you create a new application and then select **File ▶ New ▶ Other ▶ Active X ▶ Automation Server Object**.

3.2.13.23 Run

Run ▶ Run

Compiles any changed source code and, if the compile is successful, executes your application, allowing you to use and test it in the IDE .

See Also

Compiling (↗ see page 2)

3.2.13.24 Run To Cursor and Run Until Return

Run ▶ Run To Cursor

Run ▶ Run Until Return

Executes the current program you are working on and stops either at the cursor location or when the function returns.

Item	Description
Run To Cursor	Executes the current active program to the line containing the cursor. The cursor must be on a line of source code.
Run Until Return	Executes the current active program until the current procedure or function returns to its caller.

3.2.13.25 Show Execution Point

Run ▶ Show Execution Point

Opens the **Code Editor** window and bring the line of code that contains the current execution point to the front. Use this command if you closed or minimized the **Code Editor** window because the debugger automatically displays the execution point in the **Code Editor**.

3.2.13.26 Step Over

Run ▶ Step Over

Tells the debugger to execute the next line of code. If the line contains a function, Step Over executes the function and then stops at the first line after the function.

See Also

Overview of Debugging (↗ see page 10)

Using Tooltips During Debugging (↗ see page 122)

3.2.13.27 Trace Into

Run ▶ Trace Into

Tells the debugger to execute the next line of code. If the line contains a function, Trace Intro executes the function and then stops at the first line of code inside the function.

See Also

Overview of Debugging (↗ see page 10)

3.2.13.28 Trace to Next Source Line

Run ▶ Trace to Next Source Line

Executes a single source line.

3.2.13.29 Unregister ActiveX Server

Run ▶ Unregister ActiveX Server

Unregisters an "in-process" automation server from the registry.

See Also

Register ActiveX Server (⇗ see page 950)

3.2.14 Search

Topics

Name	Description
Find (⇗ see page 952)	**Search ▶ Find** Specifies the text you want to locate and sets options that affect the search. Find locates the line of code containing the first occurrence of the string and highlights it.
Find in Files (⇗ see page 953)	**Search ▶ Find in Files** Specifies the text you want to locate and sets options that affect the search. The Find In Files command works with the Repeat Search command available on the context menu of the **Messages** pane.
Find References (⇗ see page 954)	**Search ▶ Find References** Locates references to a selected identifier.
Enter Address to Position (⇗ see page 954)	**Search ▶ Goto Address** Positions to an address in the **CPU** window.
Go to Line Number (⇗ see page 954)	**Search ▶ Go to Line Number** Jumps to a line number in the **Code Editor**.
Replace Text (⇗ see page 955)	**Search ▶ Replace** Searches for specified text and then replaces with other text or with nothing.
Search Again (⇗ see page 955)	**Search ▶ Search Again** Continues to search for a specified string that was entered in Find.
Find Class (⇗ see page 955)	**Search ▶ Find Class** Opens the **Find Class** dialog box. The **Find Class** dialog box searches for all or part of a class name and lists the all classes currently in scope (from the uses and references list) that match the entered name. Select a class name from the list to open the file containing the class declaration.
Find Local References (⇗ see page 956)	**Search ▶ Find Local References** Locates references in the active code file.
Find Original Symbol (⇗ see page 956)	**Search ▶ Find Original Symbol** Searches through the list of files in the Project Manager and then displays the original declaration of the symbol in question. Select a symbol (e.g., TForm) that you are using, in another file or another section fo the project, before selecting Search for Original Symbol.
Find References (⇗ see page 956)	**Search ▶ Find References** Locates references to a selected identifier.
Incremental Search (⇗ see page 956)	**Search ▶ Incremental Search** Allows you to interactively search for text. As you type, the first matching result is highlighted in the editor. The status bar of the editor shows "Searching for:" with the text you looking for. To begin a new search, press `Backspace` to clear the status bar, or select Incremental Search again.

3.2.14.1 Find

Search ▶ Find

Specifies the text you want to locate and sets options that affect the search. Find locates the line of code containing the first occurrence of the string and highlights it.

Item	Description
Text to find	Enter a search string or use the down arrow to select a previously entered search string.
Case sensitive	Differentiates uppercase from lowercase when performing a search.
Whole words only	Searches for words only. (With this option off, the search string might be found within longer words.)
Regular expressions	Recognizes regular expressions in the search string. A list of regular expressions is given (as “special characters”) in GREP.
Forward	Searches from the current position to the end of the file. **Forward** is the default.
Backward	Searches from the current position to the beginning of the file.
Global	Searches the entire file in the direction specified by the **Direction** setting. **Global** is the default scope.
Selected text	Searches only the selected text in the direction specified by the **Direction** setting. You can use the mouse or block commands to select a block of text.
From Cursor	The search starts at the cursor's current position, and then proceeds either forward to the end of the scope, or backward to the beginning of the scope depending on the **Direction** setting. **From cursor** is the default **Origin** setting.
Entire scope	Searches the entire block of selected text or the entire file (no matter where the cursor is in the file), depending upon the **Scope** options.

3.2.14.2 Find in Files

Search ▶ Find in Files

Specifies the text you want to locate and sets options that affect the search. The Find In Files command works with the Repeat Search command available on the context menu of the **Messages** pane.

Item	Description
Case sensitive	Differentiates uppercase from lowercase when performing a search.
Whole words only	Searches for words only. If unchecked, the search string might be found within longer words.
Regular expressions	Recognizes regular expressions in the search string.
Search all files in project	Searches all files in the open project.
Search all open files	Searches files that are currently open.
Search in directories	When selected, the **Search Directory Options** are available. The search proceeds through all files indicated.
File mask	Specify the path of the files to be searched. To search other files, use a wildcard entry (such as *.* or *.txt) at the end of the path. To enter multiple masks, separate the masks with semicolons. To search for files in the product root directory, specify the root directory using the appropriate environment variable.
Include subdirectories	Searches subdirectories of the directory path specified.
Display results in separate tab in message view	Causes the results of the search to be displayed in a new search tab in the **Messages** view. The tab is labeled **Search for <string>**, where *string* is the text you searched for. If not checked, a new search tab is created unless one already exists. If a search tab exists, the search results are placed in the existing search tab and the label is changed. If no results are found, that search tab is deleted.

Tip: Each occurrence of a string is listed in the Messages

view at the bottom of the **Code Editor**. Double-click a list entry to move to that line in the code.

Tip: To repeat the last search, right-click in the Messages

view and select **Repeat Search**.

Tip: While a lengthy search is in progress, the Find in Files

command changes to **Cancel Find in Files**. To stop a search in progress, either right-click the search result tab for that search and choose **Close Tab**, or choose **Search ▶ Cancel Find in Files**.

3.2.14.3 Find References

Search ▶ Find References

Locates references to a selected identifier.

Item	Description
Remove	Red X button deletes the selected reference.
Remove All	Documents with small x icon is a button that removes all references from the **Find References** window. Successive find operations display a cumulative list of references unless you explicitly delete them.
+/-	Expands and collapses the nodes in the references tree. Each node denotes a separate file.

See Also

Finding References (see page 141)

Find References Overview (Delphi (see page 65)

3.2.14.4 Enter Address to Position

Search ▶ Goto Address

Positions to an address in the **CPU** window.

Item	Description

Edit Box	Enter the symbol, such as `main`, to which you want to position the **CPU** window.
	Alternatively, for managed code, you can enter an address in the just in time (JIT) compiler format:
	`@(module token,function token,offset)`
	For example:
	`@($3,$60005C4,$62)`
	For unmanaged code, you can enter any flat 32-bit address value, for example, $401018.

3.2.14.5 Go to Line Number

Search ▶ Go to Line Number

Jumps to a line number in the **Code Editor**.

Item	Description
Enter New Line Number	Enter the line number in the code that you want to go to, or select a number from a list of previously entered line numbers.

3.2.14.6 Replace Text

Search ▶Replace

Searches for specified text and then replaces with other text or with nothing.

Item	Description
Text to find	Enter a search string or use the down arrow to select a previously entered search string.
Replace with	Enter the replacement string. To select from a list of previously entered search strings, click the down arrow next to the input box. To replace the text with nothing, leave this input box blank.
Case sensitive	Differentiates uppercase from lowercase when performing a search.
Whole words only	Searches for words only. If unchecked, the search string might be found within longer words.
Regular expressions	Recognizes regular expressions in the search string.
Prompt on replace	Displays a confirmation prompt before replacing each occurrence of the search string. If unchecked, the **Code Editor** automatically replaces the search string.
Forward	Searches from the current position to the end of the file. **Forward** is the default.
Backward	Searches from the current position to the beginning of the file.
Global	Searches the entire file in the direction specified by the **Direction** setting. **Global** is the default scope.
Selected text	Searches only the selected text in the direction specified by the **Direction** setting. You can use the mouse or block commands to select a block of text.
From Cursor	Starts the search at the current cursor position, and then proceeds either forward to the end of the scope, or backward to the beginning of the scope depending on the **Direction** setting.
Entire scope	Searches the entire block of selected text or the entire file (no matter where the cursor is in the file), depending on the **Scope** options.
Replace All	Replaces every occurrence of the search string. If checked, the **Confirm** dialog box appears on each occurrence of the search string.

3.2.14.7 Search Again

Search ▶Search Again

Continues to search for a specified string that was entered in Find.

3.2.14.8 Find Class

Search ▶Find Class

Opens the **Find Class** dialog box. The **Find Class** dialog box searches for all or part of a class name and lists the all classes currently in scope (from the uses and references list) that match the entered name. Select a class name from the list to open the file containing the class declaration.

3.2.14.9 Find Local References

Search ▶ Find Local References

Locates references in the active code file.

See Also

Finding References (see page 141)

findrefov.xml (see page 65)

3.2.14.10 Find Original Symbol

Search ▶ Find Original Symbol

Searches through the list of files in the Project Manager and then displays the original declaration of the symbol in question.

Select a symbol (e.g., TForm) that you are using, in another file or another section fo the project, before selecting Search for Original Symbol.

See Also

Find References Overview (Delphi (see page 65)

3.2.14.11 Find References

Search ▶ Find References

Locates references to a selected identifier.

Item	Description
Remove	Red X button deletes the selected reference.
Remove All	Documents with small x icon is a button that removes all references from the **Find References** window. Successive find operations display a cumulative list of references unless you explicitly delete them.
+/-	Expands and collapses the nodes in the references tree. Each node denotes a separate file.

See Also

Finding References (see page 141)

Find References Overview (Delphi (see page 65)

3.2.14.12 Incremental Search

Search ▶ Incremental Search

Allows you to interactively search for text. As you type, the first matching result is highlighted in the editor. The status bar of the editor shows "Searching for:" with the text you looking for.

To begin a new search, press `Backspace` to clear the status bar, or select Incremental Search again.

3.2.15 Together

Topics

Name	Description
Add New Diagram dialog box (↗ see page 960)	**Context menu ▶ Add ▶ Other Diagram** To open this dialog box, right click the project root or a namespace element in the **Diagram View** or **Model View**, and choose **Add ▶ Other Diagram** on the context menu. Alternatively, with the project root selected or a namespace element selected, you can use the keyboard shortcut, `CTRL+SHIFT+D` to open the dialog box. This dialog box displays the available diagrams that you can add to your project.
Add/Remove Parameters for Operation dialog box (↗ see page 961)	To open this dialog box, select a function or procedure on a diagram in the **Model View** or **Diagram View**, and click the elipsis at the right of the **Params** property in the **Object Inspector**. This dialog box is used to specify parameters for functions or procedures. You can add, edit, and delete entries.
Add/Remove User Properties dialog box (↗ see page 961)	**Context menu ▶ User Properties** To open this dialog, right-click the diagram or model element in the **Model View** or **Diagram View**, and choose **User Properties** on the context menu. This dialog box is used for creating user properties and OCL constraints. The dialog box displays a list of properties, if any, each entry consisting of a pair Name-Value. You can add and delete entries, and edit names and values. For editing names, use the **Name** text field. For editing values, you can either use the text field, or click the button and enter the text in the editor widow.... more (↗ see page 961)
Change Parameters dialog box (↗ see page 961)	**Refactor ▶ Change Parameters** You can open the **Change Parameters** dialog box on the Refactoring main menu, or by using the **Refactoring ▶ Change Parameters** command on the context menu for methods. **Note:** This feature is available for implementation projects only.
Choose Destination (or: Source) dialog box (↗ see page 962)	**Model View context menu ▶ Transform to Source (or: Transform Code from Design Project)** This dialog box is invoked from the context menu of a project in the **Model View**. Select the desired target project and click **Transform**.
Diagram Layout Algorithms (↗ see page 963)	The following diagram layout algorithms are available: • **Autoselect**: several algorithms can be available for each diagram type. This option analyzes internal information of each algorithm, and selects the one that best suits the current diagram type. If autoselect: Each of the layout algorithms contains internal information about the types of diagrams it will work with and the numeric characteristics for the final quality of the produced layout when applied to each applicable diagram type. Several algorithms can be available for the same diagram type. The autoselect option uses such internal information and picks the best algorithm for the current... more (↗ see page 963)
Edit Hyperlinks for Diagram dialog box (↗ see page 963)	**Context menu ▶ Hyperlinks ▶ Edit** This dialog box is invoked from the context menus in the Diagram Editor or the **Model View**. This dialog box contains two tabbed pages that enable you to create hyperlinks to the model elements and external resources.
Export Diagram to Image dialog box (↗ see page 964)	**File ▶ Export Diagram to Image** This dialog box allows you to save the active diagram in the specified format. To open this dialog box, place the focus on the diagram to export in the **Diagram View** and choose [File | Export Diagram to Image...].

Extract Interface or Superclass dialog box (↗ see page 964)	**Refactor ▶ Extract Superclass (or: Extract Interface)** You can open the Extract Interface/Superclass dialogs from the Refactoring main menu, or by using the Refactoring | Extract Superclass (or Extract Interface) commands on the context menu of applicable class diagram elements. The Extract Interface command is available for classes, structures, methods, properties, events, and indexers. The Extract Superclass command is available for classes, interfaces (Extract Superinterface), methods, properties, events, fields, and indexers. **Note:** This feature is available for implementation projects.
Extract Method dialog box (↗ see page 965)	**Refactor ▶ Extract Method** You can open the **Extract Method** dialog box from the Refactoring main menu, or by using the **Refactoring ▶ Extract Method** command on several complete statements in the RAD Studio Editor. **Note:** This feature is available for implementation projects.
Generate Documentation dialog box (↗ see page 966)	**Tools ▶ Options ▶ Together ▶ Generate Documentation** Together features a UML documentation wizard that you can use to generate HTML documentation for your projects. To open this dialog, choose [Tools | Generate Documentation...] from the main menu. The Together dialog boxes have built-in help, in addition to this help.
Generate Sequence Diagram dialog box (↗ see page 966)	**Context menu ▶ Generate Sequence Diagram** To open this dialog box, right-click a method (or function) and choose Generate Sequence Diagram from the context menu. The Generate Sequence Diagram dialog box lists the classes and namespaces involved in the method (function) and allows you to choose which classes/namespaces to display on the generated sequence diagram.
Inline Variable dialog box (↗ see page 967)	**Refactor ▶ Inline Variable** The dialog box reports the number of variable occurrences that the Inline Variable command will be applied to. Click OK to complete the changes. You can open the Inline Variable dialog box from the Refactoring main menu, or by using the Refactoring | Inline Variable command on a local variable in the Delphi Code EditorVisual Studio Editor. **Note:** The variable that you select should not be updated later in the source code. If it is, the error message "Variable index is accessed for writing." opens. **Note:** This feature is available for implementation projects only. **Note:... more (↗ see page 967)**
Introduce Field dialog box (↗ see page 968)	**Refactor ▶ Introduce Field** You can open the Introduce Field dialog from the Refactoring main menu, or by using the Refactoring | Introduce Field command on an expression in the Delphi Code EditorVisual Studio Editor. **Note:** This feature is available for implementation projects only. **Note:** This command is only available while working in the Delphi Code Editor Visual Studio Editor.
Introduce Variable dialog box (↗ see page 968)	**Refactor ▶ Introduce Variable** You can open the Introduce Variable dialog from the Refactoring main menu, or by using the Refactoring | Introduce Variable command on a variable in the Code Editor. **Note:** This feature is available for implementation projects only. **Note:** This command is only available while working in the Code Editor.
Model Support (↗ see page 969)	**Project ▶ Together Support** This dialog box lets you enable or disable Together modeling support for the currently opened projects.
Move dialog box (↗ see page 969)	**Refactor ▶ Move** The Move dialog opens when you choose the Move command from the Refactoring menu, or by using the Refactoring | Move command on the context menu of static methods, static fields, and static properties (collectively, static members). **Note:** This feature is available for implementation projects only.

Together Options dialog window (see page 969)	**Tools ▶ Options ▶ Together** The **Options** dialog window displays a tree view of configuration option categories, each of which displays a set of individual configuration options when selected. To open this dialog box, choose **Tools ▶ Options** on the main menu. Select the Together folder from the tree view list on the left of the dialog window. The Together dialog boxes have built-in help, in addition to this help. The help for a selected option is displayed at the bottom of the dialog window. The following option categories exist in the tree view under the configuration levels:
Print Audit dialog box (see page 970)	**Audit results pane ▶ Print button** This dialog box enables you to print selected sets of audit report results to the specified printer. The dialog box is invoked from the audit results report view.
Print Diagram dialog box (see page 971)	**File ▶ Print** This dialog box enables you to print selected diagrams to the specified printer. The dialog box is invoked by choosing File | Print from the main menu with a diagram open in the **Diagram View**.
Pull Members Up and Push Members Down dialog boxes (see page 972)	**Refactor ▶ Pull Members Up (or: Push Members Down)** You can open the Pull Members Up/Push Members Down dialog boxes from the Refactoring main menu, or by using the Refactoring | Pull Members Up (or Push Members Down) commands on the context menu of applicable class diagram elements. Both the Pull Members Up/Push Members Down commands are available for methods, properties, fields, indexers, and events. **Warning:** This feature is available for implementation projects only.
QA Audits dialog window (see page 972)	**Context menu ▶ QA Audits** Open the Audits dialog window by choosing Tools | Together | QA Audits from the main menu or by choosing QA Audits from the **Diagram View**, **Model View**, or class/interface context menus. **Warning:** This feature is available for implementation projects only.
QA Metrics dialog window (see page 974)	**Context menu ▶ QA Metrics** Open the Metrics dialog by choosing [Tools | Together | QA Metrics...] from the main menu, or by choosing QA Metrics from the **Diagram View**, **Model View**, or class/interface context menus. **Warning:** This feature is available for implementation projects only.
Rename (see page 975)	**Refactor ▶ Rename** Opens when you choose the Rename command from the Refactoring menu, or by using the Refactoring | Rename command on the context menu of code-generating class diagram elements. Renaming is applicable to classes, interfaces, enumerators, structures, delegates, methods, properties, events, and fields. **Warning:** This feature is available for implementation projects only.
Safe Delete dialog box (see page 975)	**Refactor ▶ Safe Delete** You can open the Safe Delete dialog box from the Refactoring main menu, or by using the Refactoring | Safe Delete command on the context menu of applicable class diagram elements. The Safe Delete command is available for all code-generating class diagram elements. It is not available for namespace elements. **Warning:** This feature is available for implementation projects only.
Save Audit and Metric Results dialog box (see page 976)	**QA Audits (or: Metrics) pane ▶ Save button** This dialog box is invoked from the audit or metrics results report on pressing the Save button. **Note:** This feature is available in implementation projects only.
Search for Usages dialog box (see page 977)	**Search ▶ Search for Usages** This dialog box provides a flexible tool to track references to, and overrides of, elements and members in the source-code projects.

Select element dialog box (see page 977)	This dialog box displays a tree view of the available contents within your project groupsolution. Expand the project nodes to reveal the nested classes, select the required element, and click OK when ready. This dialog box belongs to a general group of selection dialogs where you can choose interactions, operations, ancestor classes, instantiated classes for the objects, etc. This dialog opens when you press the chooser button in a field of the Object InspectorProperties Window, or when More is selected from the Choose Class or Choose Method menu nodes.
Selection Manager (see page 978)	This dialog belongs to a general group of selection dialogs where you can select elements from the available contents and add them to a certain destination scope. All Selection Manager dialogs have a similar structure and varying title strings.
XMI Export dialog box (see page 978)	**File ▶ Export Project to XMI** Use this dialog box to export a Together model to an XML file with the model described in XMI. To open this dialog box, choose File | Export Project to XMI from the main menu while the project root node is selected in the **Model View**. You can also right-click the project root node in the **Model View** and choose Export Project to XMI from the context menu.
XMI Import dialog box (see page 979)	**File ▶ Import Project from XMI** Use this dialog box to import an XML file with the model described in XMI. To open this dialog box, choose File | Import Project from XMI from the main menu while the project root node is selected in the **Model View**. You can also right-click the project root node in the **Model View**, and choose Import Project from XMI from the context menu.

3.2.15.1 Add New Diagram dialog box

Context menu ▶ Add ▶ Other Diagram

To open this dialog box, right click the project root or a namespace element in the **Diagram View** or **Model View**, and choose **Add ▶ Other Diagram** on the context menu. Alternatively, with the project root selected or a namespace element selected, you can use the keyboard shortcut, `CTRL+SHIFT+D` to open the dialog box.

This dialog box displays the available diagrams that you can add to your project.

Item	Description
Diagrams tab	The Diagrams tab lists the available UML diagram types. You can change the view of the UML diagram icons by pressing the Small Icons or Large Icons buttons. By default, large icons display in the Diagrams tab.
Diagram name	By default, your selected diagram type is displayed in the Name field. You can edit its name or enter a new name for the new diagram. This name is displayed in the **Model View** and in the diagram's tab in the **Diagram View** when the diagram is open for editing.
Buttons	
Large Icons	The default setting. This button controls the appearance of the UML diagram icons. Large UML diagram icons display in the dialog box when selecting this button.
Small Icons	This button controls the appearance of the UML diagram icons. Small UML diagram icons display in the dialog box when selecting this button.
OK	Creates the new diagram of the selected type, opens it in the **Diagram View** on a new tab, and closes the Add New Diagram dialog box.
Cancel	Discards all changes and closes the Add New Diagram dialog box.
Help	Displays this help topic.

See Also

Diagram Overview (see page 90)

Creating a Diagram (see page 196)

3.2.15.2 Add/Remove Parameters for Operation dialog box

To open this dialog box, select a function or procedure on a diagram in the **Model View** or **Diagram View**, and click the elipsis at the right of the **Params** property in the **Object Inspector**.

This dialog box is used to specify parameters for functions or procedures. You can add, edit, and delete entries.

Add	Creates a new entry in the list of parameters.
Remove	Deletes the selected entry from the list of parameters.
OK	Saves changes and closes the dialog box.
Cancel	Discards changes and closes the dialog box.
Help	Displays this topic.

3.2.15.3 Add/Remove User Properties dialog box

Context menu ▶ User Properties

To open this dialog, right-click the diagram or model element in the **Model View** or **Diagram View**, and choose **User Properties** on the context menu.

This dialog box is used for creating user properties and OCL constraints. The dialog box displays a list of properties, if any, each entry consisting of a pair Name-Value.

You can add and delete entries, and edit names and values. For editing names, use the **Name** text field. For editing values, you can either use the text field, or click the button and enter the text in the editor widow.

Add	Creates a new entry in the list of properties.
Remove	Deletes the selected entry from the list of properties.
OK	Saves changes and closes the dialog box.
Cancel	Discards changes and closes the dialog box.
Help	Displays this topic.

See Also

OCL support overview (see page 95)

Working with user properties (see page 206)

3.2.15.4 Change Parameters dialog box

Refactor ▶ Change Parameters

You can open the **Change Parameters** dialog box on the Refactoring main menu, or by using the **Refactoring ▶ Change Parameters** command on the context menu for methods.

Note: This feature is available for implementation projects only.

Class	A read-only field displaying the name of the class where the method resides.
Method	A read-only field displaying the selected member and its current parameters, if applicable.
Select members	A table displays all existing parameters and any new parameters that you add to the method. The order of the parameters in the table is the order of the parameters in the method. Use the Add and Remove buttons to add and remove parameters from the method. If adding a new parameter, you can edit its Type, Name, and Default value. If editing an existing parameter, you can edit its Name. You can rearrange the order of the parameters using the Move Up and Move Down buttons.
Preview Usages	By default, **Preview Usages** is checked. If this option is checked when you click OK, the **Refactoring** window opens allowing you to review the refactoring before committing to it. If this option is cleared when you click OK, the **Refactoring** window opens with the change parameters operation completed.
Buttons	
Add	Adds a new parameter to the method.
Remove	Removes the currently-selected parameter from the method.
Move Up	Moves the currently-selected parameter up one row.
Move Down	Moves the currently-selected parameter down one row.
OK	Opens the Refactoring window.
Cancel	Discards all changes and closes the dialog box.

See Also

Refactoring overview (see page 98)

Changing parameters in methods (see page 184)

3.2.15.5 Choose Destination (or: Source) dialog box

Model View context menu ▶ Transform to Source (or: Transform Code from Design Project)

This dialog box is invoked from the context menu of a project in the **Model View**. Select the desired target project and click **Transform**.

Item	Description
Existing projects	Displays the list within the current project groupsolution. For implementation projects, the design projects are greyed out. For design projects, the implementation projects are greyed out.
Use name mapping files for code generation	This checkbox enables or disables support of the **name mapping** feature.
Transform	Press this button to start transforming design project to source code. Note that the button is only enabled when a valid source-code project is selected in the list.
Cancel	Press this button to discard selection and close the dialog box.
Help	Opens this topic.

See Also

Transformation to source code overview (see page 94)

Transforming design project to source code (see page 269)

3.2.15.6 Diagram Layout Algorithms

The following diagram layout algorithms are available:

- **Autoselect**: several algorithms can be available for each diagram type. This option analyzes internal information of each algorithm, and selects the one that best suits the current diagram type. If autoselect: Each of the layout algorithms contains internal information about the types of diagrams it will work with and the numeric characteristics for the final quality of the produced layout when applied to each applicable diagram type. Several algorithms can be available for the same diagram type. The autoselect option uses such internal information and picks the best algorithm for the current diagram type.
- **Hierarchical**: this type of algorithm is most suitable to analyze hierarchical structure (for example study inheritance relationships). The Hierarchical algorithm originates from the Sugiyama algorithm. The algorithm draws the UML diagram hierarchically according to the preferences that you select.
- **Together**: algorithm applicable to all types of diagrams. It includes the layout options used in version 6.1 of Together ControlCenter and Together Edition for JBuilder.
- **Tree**: the algorithm draws a tree diagram in a tree layout. The algorithm draws the given graph in a tree layout according to its maximum spanning tree.
- **Orthogonal**: simple structural algorithm is used when hierarchy is not important. The Orthogonal algorithm uses heuristics to distribute diagram nodes among a lattice.
- **Spring Embedder**: Spring Embedder are force-directed layout algorithms that model the input graph as a system of forces and try to find a minimum energy configuration of this system. All edges are drawn as straight lines. This type of layout is especially suitable for projects with numerous diagram elements based on large amount of source code. When you lay out a graph according to the Spring Embedder layout algorithm, the program will simulate the graph as a physical model (masses and springs) and subject it to physical forces. The unnecessarily-long edges will be the most tense, and will try to contract the most. When the nodes and edges have been balanced, you will have a geometric representation of the graph.

See Also

Diagram Layout Overview (see page 92)

Laying Out a Diagram Automatically (see page 202)

3.2.15.7 Edit Hyperlinks for Diagram dialog box

Context menu ▶ Hyperlinks ▶ Edit

This dialog box is invoked from the context menus in the Diagram Editor or the **Model View**. This dialog box contains two tabbed pages that enable you to create hyperlinks to the model elements and external resources.

Dialog title	The title of the dialog box varies depending on the way it is invoked. It displays the string that corresponds to the invoking object.
Model Elements tab	The pane on the left of the dialog box displays the content available in your project. You can use the explorer to navigate to the element and select it for inclusion in the pane of values returned by the dialog to the invoking object.
External Documents tab	The Recently Used Documents pane displays the external contents that you make available for your project. Such contents may be represented by the file system resources or by URLs. Use the Browse and URL buttons to specify these resources.
Browse	Click this button to invoke the Open dialog box. Navigate to the desired file and click OK.
URL	Click this button to invoke the Documents URL dialog box. Type a URL in the text field and click OK.
Clear	Click this button to remove all entries in the list of the Recently Used Documents.

Selected pane	This pane displays two kinds of data: Values already existing and passed from the invoking object, if any. Values of the selections you have added from the left-hand pane, if any.

Buttons	
Add	Enabled when an element is selected in the left-hand pane. Adds the selected element to the right-hand pane.
Remove	Enabled when you select an item in the right-hand pane. Removes the selected item from the pane. All removed values or objects are removed from the invoking property or diagram upon clicking OK.
Remove All	Enabled when items are present in the right-hand pane. Removes all items from that pane. All removed values or objects are removed from the invoking property or diagram upon clicking OK.

See Also

Hyperlinking overview (see page 92)

Hyperlinking diagrams (see page 201)

3.2.15.8 Export Diagram to Image dialog box

File ▶ Export Diagram to Image

This dialog box allows you to save the active diagram in the specified format. To open this dialog box, place the focus on the diagram to export in the **Diagram View** and choose [File | Export Diagram to Image...].

Zoom	Use this section to specify the zoom factor and dimensions of the image.
Z	Enter a zoom factor.
W	Enter the image width in pixels.
H	Enter the image height in pixels.
Preview	Click the down arrow to show the print preview page.
Preview zoom	Use the slider to set up the preview zoom. The current value of the zoom factor is displayed to the left of the slider.
Auto preview zoom	Check this option to fit the image to the preview window.
Save	Click this button to open the Save dialog box. Specify the target file name, target location, and the format of the exported image.

See Also

Export and import features (see page 100)

Exporting diagram to image (see page 197)

3.2.15.9 Extract Interface or Superclass dialog box

Refactor ▶ Extract Superclass (or: Extract Interface)

You can open the Extract Interface/Superclass dialogs from the Refactoring main menu, or by using the Refactoring | Extract Superclass (or Extract Interface) commands on the context menu of applicable class diagram elements. The Extract Interface command is available for classes, structures, methods, properties, events, and indexers. The Extract Superclass command is available for classes, interfaces (Extract Superinterface), methods, properties, events, fields, and indexers.

Note: This feature is available for implementation projects.

Interface/Superclass name	Enter the name of the interface or superclass to be created.
Namespace	Use this field to specify a namespace where the interface/superclass will reside. You must enter a fully-qualified name for the namespace. Alternatively, press the button and select the desired target namespace.
Select members	A table displays the detected members that you can choose to extract to the new interface or superclass. By default, all detected members are selected. Use the checkboxes in the first column of the table to indicate which members to extract.
View references before refactoring	By default, View references before refactoring is checked. If this option is checked when you click OK, the Refactoring window opens allowing you to review the refactoring before committing to it. If this option is cleared when you click OK, the Refactoring window opens with the extraction completed.
Buttons	
OK	Opens the Refactoring window.
Cancel	Discards all changes and closes the dialog.

See Also

Refactoring overview (↗ see page 98)

Extracting interfaces and superclasses (↗ see page 185)

3.2.15.10 Extract Method dialog box

Refactor ▸ Extract Method

You can open the **Extract Method** dialog box from the Refactoring main menu, or by using the **Refactoring ▸ Extract Method** command on several complete statements in the RAD Studio Editor.

Note: This feature is available for implementation projects.

Note	This command is only available while working in the RAD Studio Editor.
Name	Enter the name of the method to be created from the selected fragment of code.
Visibility	Choose a visibility modifier from the drop-down list. Your choices are: public, protected, private, internal, internal protected.
Header comment	Enter a code comment describing the new method.
Static	Check the checkbox to set the **Static** field, if necessary.

See Also

Refactoring Overview (↗ see page 98)

Extracting Methods (↗ see page 185)

3.2.15.11 Generate Documentation dialog box

Tools ▶ Options ▶ Together ▶ Generate Documentation

Together features a UML documentation wizard that you can use to generate HTML documentation for your projects. To open this dialog, choose [Tools | Generate Documentation...] from the main menu. The Together dialog boxes have built-in help, in addition to this help.

Scope options	You can limit the scope of the documentation to a smaller set by choosing a different Scope option. The Scope section at the top of the dialog has radio buttons to indicate what parts of the project should be parsed and included in the generated documentation:
Current namespace	Generated output includes only the current namespace selected in the **Model View** or in the **Diagram View**.
Current namespace with descendent namespaces	Generated output includes the current namespace selected in the Model View and any descendent namespaces under it.
Current diagram	Generated output for the current diagram that is in focus in the Diagram View.
All	Generated output covers the entire project.
Options settings	The Options section of the dialog has options to specify the destination and other optional actions:
Output folder	Enter the location for the generated output, or select from the file chooser.
Include diagrams	Check to include diagram images in the output.
Include navigation tree	Check to include a navigation tree in the output.
Launch HTML browser	Check to load the documentation into your external web browser.
Note	The navigation frame in the documentation will work only if JDK/JRE 1.4 is installed and enabled in your browser.
Buttons	
OK	Accepts the input and starts the generate documentation process.
Cancel	Cancels your input and closes the dialog box without generating documentation.

See Also

Generating project documentation overview (see page 100)

Generating project documentation (see page 248)

3.2.15.12 Generate Sequence Diagram dialog box

Context menu ▶ Generate Sequence Diagram

To open this dialog box, right-click a method (or function) and choose Generate Sequence Diagram from the context menu. The Generate Sequence Diagram dialog box lists the classes and namespaces involved in the method (function) and allows you to choose which classes/namespaces to display on the generated sequence diagram.

Fields	
Name	Lists the names of namespaces/classes involved in the method (function).
Show On Diagram	Check the namespaces/classes that you want to show on the generated sequence diagram. All namespaces and classes are selected by default. However, some classes may not be relevant. To increase the meaningfulness of the generated diagram, clear the checkboxes that are not helpful in explaining the sequence of operations.
Show Implementation	For the elements that you decide to show in the diagram, check whether to show the implementation details in the generated sequence diagram.
Buttons	
OK	Generates the new sequence diagram and opens the diagram in a new tab in the **Diagram View**.
Cancel	Closes the dialog box without generating a sequence diagram.
Help	Displays this help topic.

See Also

Roundtrip engineering overview

Options for sequence diagram generation (↗ see page 1102)

3.2.15.13 Inline Variable dialog box

Refactor ▸ Inline Variable

The dialog box reports the number of variable occurrences that the Inline Variable command will be applied to. Click OK to complete the changes.

You can open the Inline Variable dialog box from the Refactoring main menu, or by using the Refactoring | Inline Variable command on a local variable in the Delphi Code EditorVisual Studio Editor.

Note: The variable that you select should not be updated later in the source code. If it is, the error message "Variable index is accessed for writing." opens.

Note: This feature is available for implementation projects only.

Note: This command is only available while working in the Delphi Code Editor

Visual Studio Editor, not in the **Diagram View**.

Buttons	
OK	Creates the inline variable and closes the dialog box.
Cancel	Discards all changes and closes the dialog box.

See Also

Refactoring overview (↗ see page 98)

Creating inline variables (↗ see page 186)

3.2.15.14 Introduce Field dialog box

Refactor ▶ Introduce Field

You can open the Introduce Field dialog from the Refactoring main menu, or by using the Refactoring | Introduce Field command on an expression in the Delphi Code EditorVisual Studio Editor.

Note: This feature is available for implementation projects only.

Note: This command is only available while working in the Delphi Code Editor

Visual Studio Editor.

Name	Enter a name for the new field.
Visibility	Choose the visibility for the new field. Using the combo box, choose from public, protected, private, internal, or internal protected.
Initialize	Choose where to initialize the new field. Using the combo box, choose from Current method, Class constructor(s), or Field declaration.
Static	If applicable, check the Static field.
Replace all occurrences	If applicable, check this field to replace all occurrences of the expression.
Buttons	
OK	Creates the new field and closes the dialog box.
Cancel	Discards all changes and closes the dialog box.

See Also

Refactoring overview (see page 98)

Introducing Fields (see page 187)

3.2.15.15 Introduce Variable dialog box

Refactor ▶ Introduce Variable

You can open the Introduce Variable dialog from the Refactoring main menu, or by using the Refactoring | Introduce Variable command on a variable in the Code Editor.

Note: This feature is available for implementation projects only.

Note: This command is only available while working in the Code Editor.

Name	Enter a name for the new variable. The variable created is given the same type as the original variable.
Replace all occurrences	If applicable, check this field to replace all occurrences of the expression. The Introduce Variable dialog indicates the number of occurrences that it will replace with the new variable. Note that the refactoring does not replace any occurrences of the variable prior to the point in the code at which you selected to introduce the new variable.
Buttons	
OK	Creates the new variable and closes the dialog box.
Cancel	Discards all changes and closes the dialog box.

See Also

Refactoring overview (see page 98)

Introducing new variables (see page 187)

3.2.15.16 Model Support

Project▶Together Support

This dialog box lets you enable or disable Together modeling support for the currently opened projects.

Item	Description
Project list	Displays the project in the current project groupsolution.

See Also

Activating Together Support for Projects (see page 263)

3.2.15.17 Move dialog box

Refactor▶Move

The Move dialog opens when you choose the Move command from the Refactoring menu, or by using the Refactoring | Move command on the context menu of static methods, static fields, and static properties (collectively, static members).

Note: This feature is available for implementation projects only.

Move Members	This field displays the list of selected static members. You can move more than one static member at a time. Deselect/select the static members by clearing/checking the check box next to the name of a member.
To (namespace fully qualified name)	Use this field to select a class where the static member or members will reside. You must enter a fully-qualified name for the class or click the browse button to select one.
Preview Usages	By default, Preview Usages is checked. If this option is checked when you click OK, the Refactoring window opens allowing you to review the refactoring before committing to it. If this option is cleared when you click OK, the Refactoring window opens and the move is completed.
Buttons	
OK	Opens the Refactoring window.
Cancel	Discards all changes and closes the dialog box.

See Also

Refactoring overview (see page 98)

Moving code elements (see page 188)

3.2.15.18 Together Options dialog window

Tools▶Options▶Together

The **Options** dialog window displays a tree view of configuration option categories, each of which displays a set of individual configuration options when selected. To open this dialog box, choose **Tools ▶ Options** on the main menu. Select the Together folder from the tree view list on the left of the dialog window.

The Together dialog boxes have built-in help, in addition to this help. The help for a selected option is displayed at the bottom of the dialog window.

The following option categories exist in the tree view under the configuration levels:

General	The General options allow you to customize certain behaviors in the user interface that do not pertain to any other specific category of options such as Diagram.
Diagram	The Diagram options control a number of default behaviors and appearances of diagrams: **Appearance**, **Layout**, **Print**, and **View Management**.
Generate documentation	The Generate Documentation options control the variety of content (as well as appearance) to include or exclude from your generated HTML documentation.
Model View	The Model View options control how diagram content displays in the **Model View**.
Sequence diagram roundtrip	The Sequence Diagram Roundtrip options apply to generating sequence diagrams from source code and generating source code from a sequence diagram.
Source code	The Source Code options allows you to control several LiveSource parameters.

Buttons	
OK	Applies changes, and closes the dialog window.
Cancel	Closes the dialog window without saving any changes.
Help	Displays the RAD Studio online help.

See Also

Configuring Together (see page 183)

Configuration levels (see page 1088)

Option value editors (see page 1101)

General options (see page 1098)

Diagram Appearance options (see page 1089)

Diagram Layout options (see page 1091)

Diagram Print options (see page 1094)

Diagram View Filters options (see page 1096)

Generate Documentation options (see page 1099)

Model View options (see page 1101)

Sequence Diagram Roundtrip options (see page 1102)

Source Code options (see page 1103)

3.2.15.19 Print Audit dialog box

Audit results pane ▶ Print button

This dialog box enables you to print selected sets of audit report results to the specified printer. The dialog box is invoked from the audit results report view.

Select View	Choose the scope of the results to print using the Select View list box. Audit results display in tabbed-pages in the audit results report view. You can group and ungroup the results using the Group by command on the report view context menu. Unless the results have been grouped using the Group by command, the Active Group option is not enabled in the dialog. The possible view options are: **All Results**: If the results are grouped, choosing All Results prints a report for all groups in the current tabbed-page. If the results are not grouped, then all results print for the current tabbed-page. **Active Group** : If the results are grouped, you can select a group in the current tabbed page to print a report for the selected group. **Selected Rows**: You can select single or multiple rows in the audit results report view. Choosing Selected Rows prints a report for such selections.
Print zoom	Type in a zoom factor for the printout. By default, the zoom factor is set to 1.
Fit to page	Check this option if you want to print the results on a single page. If checked, the Print zoom field is disabled.
Preview	Click the down arrow to show the print preview page.
Preview zoom	Use the Preview zoom (auto) slider to set the preview zoom. The current value of the zoom factor is displayed to the left of the slider.
Auto preview zoom	Check this option to fit the image to the preview window.
Buttons	
Print	Click Print to send the selected audits report to the default printer. Use the down arrow to choose the Print dialog command, which enables you to configure the printer options.
Cancel	Click to close the dialog box without printing the audits report.
Help	Clicking Help opens this page.

See Also

Viewing audit results (see page 274)

Print Audit dialog box

Audit results pane (see page 1111)

3.2.15.20 Print Diagram dialog box

File ▶ Print

This dialog box enables you to print selected diagrams to the specified printer. The dialog box is invoked by choosing File | Print from the main menu with a diagram open in the **Diagram View**.

Print diagrams	From this list box, choose the diagrams to be printed. The possible options are: Active diagram Active with neighbors (all diagrams within the same namespace) All opened diagrams All diagrams in the model
Print zoom	Enter a zoom factor for the printout. By default, the zoom factor is set to 1.
Fit to page	Check this option if you want to print the diagram on a single page. If checked, the Print zoom field is disabled.

Preview	Click the down arrow to show the print preview page.
Preview zoom	Use the slider to set up the preview zoom. The current value of the zoom factor is displayed to the left of the slider.
Auto preview zoom	Check this option to fit the image to the preview window.
Print	Press this button to send the selected diagrams to the default printer. Use the down arrow to choose the Print dialog box command, which enables you to configure the printer options.

See Also

Printing a diagram (see page 197)

3.2.15.21 Pull Members Up and Push Members Down dialog boxes

Refactor ▶ Pull Members Up (or: Push Members Down)

You can open the Pull Members Up/Push Members Down dialog boxes from the Refactoring main menu, or by using the Refactoring | Pull Members Up (or Push Members Down) commands on the context menu of applicable class diagram elements. Both the Pull Members Up/Push Members Down commands are available for methods, properties, fields, indexers, and events.

Warning: This feature is available for implementation projects only.

Select members:	A table displays the selected members that you have chosen to pull up (or push down). By default, all members are selected. Use the checkboxes in the first column of the table to indicate which members to pull up/push down. The third column allows you to indicate whether to make the member abstract.
Select the class to pull (or push) member to:	At the bottom of the dialog, a hierarchy displays. Select the class where you want to pull up/push down your selected members.
View references before refactoring:	By default, View references before refactoring is checked. If this options is checked when you click OK, the Refactoring window opens allowing you to review the refactoring before committing to it. If this option is cleared when you click OK, the Refactoring window opens with the pull up/push down operation completed.
Buttons	
OK	Opens the Refactoring window.
Cancel	Discards all changes and closes the dialog box.

See Also

Refactoring overview (see page 98)

Pulling up and pushing down members (see page 188)

3.2.15.22 QA Audits dialog window

Context menu ▶ QA Audits

Open the Audits dialog window by choosing Tools | Together | QA Audits from the main menu or by choosing QA Audits from the **Diagram View**, **Model View**, or class/interface context menus.

Warning: This feature is available for implementation projects only.

Toolbar	Use the Toolbar on the Audits dialog to load and save custom audit sets and to specify which audits to run on your projects. The Toolbar buttons are descibed in the table below.
Toolbar button	Description
Load Set	Opens a file chooser for selecting an .adt file to load a custom set of audits.
Save Set As	Opens a Save dialog for specifying the name and location to save the currently selected set of audits as an .adt file.
Select All	Checks the boxes for all audits in the dialog.
Unselect All	Unchecks the boxes for all audits in the dialog box.
Set Defaults	Resets the selected audits to correspond to the default set (saved in the default.adt file).
Find Audit	Navigates to the audit whose name starts with the specified string.
Scope	This field indicates the parts of the project that the Audits will be run against. Use the drop down arrow to set the Scope for either the current Selection or for the entire Model. When choosing Selection, audits are processed only for the diagram, namespace, or class that you selected before invoking the Audits dialog. If you choose Model, audits are processed for the entire project. Tip: If you have not selected any items in the Diagram or **Model View**, the Scope option defaults to the entire project. If you want to run audits on specific classes, namespaces, or diagrams, make sure you correctly select them before you open the Audits dialog.
Selection pane	The selection pane provides a list of available audits, organized by category. Check the boxes for the audits that you want to run. Check or clear the box for a category to select or unselect all of the audits in the category. As you click on an audit in the selection pane, its corresponding description displays in the lower pane of the dialog window. The descriptions include brief explanations of what the audit looks for, examples of violations, and advice on how to correct the code.
Options pane	The set of options vary depending on the selected audit. Where necessary, option controls are explained in the description for the particular audit. Use the toolbar buttons to display properties in the desired manner. Other options are displayed when applicable to the selected audit.
Button	Description
Categorized	Displays the properties of the audit in expandable groups.
Alphabetic	Displays the properties of the audit in alphabetical order.
Severity	Severity is always present; you can assign an Info, Warning, Error, or Fatal level of severity to each selected audit. The severity level defines how serious the violations are. The selected severity level is displayed in the results.
Buttons	
Start	Runs the selected set of audits.
Cancel	Discards all changes and closes the dialog box.
Help	Opens this Help topic.

See Also

Quality Assurance facilities overview (see page 98)

Running audits (see page 273)

Viewing audit results (see page 274)

3.2.15.23 QA Metrics dialog window

Context menu ▶ QA Metrics

Open the Metrics dialog by choosing [Tools | Together | QA Metrics...] from the main menu, or by choosing QA Metrics from the **Diagram View**, **Model View**, or class/interface context menus.

Warning: This feature is available for implementation projects only.

Toolbar	Use the Toolbar on the Metrics dialog window to load and save custom metrics sets and to specify which metrics to run on your projects. The Toolbar buttons are described in the table below.
Button	Description
Load Set	Opens a file chooser for selecting a .mts file to load a custom set of metrics.
Save Set As	Opens a Save dialog for specifying the name and location to save the currently-selected set of metrics as a .mts file.
Select All	Checks the boxes for all metrics in the dialog.
Unselect All	Unchecks the boxes for all metrics in the dialog.
Set Defaults	Resets the selected metrics to correspond to the default set (saved in the default.mts file).
Find Metrics	Navigates to the metric whose name starts with the specified string.
Scope	This field indicates the parts of the project that the Metrics will be run against. Use the drop-down arrow to set the Scope for either the current Selection or for the entire Model. When choosing Selection, Metrics are processed only for the diagram, namespace, or class that you selected before invoking the Metrics dialog. If you choose Model, Metrics are processed for the entire project. Tip: If you have not selected any items in the Diagram or **Model View**, the Scope option defaults to the entire project. If you want to run Metrics on specific classes, namespaces, or diagrams, make sure you correctly select them before you open the Metrics dialog.
Selection pane	The selection pane provides a list of available Metrics, organized by category. Check the boxes for the Metrics that you want to run. Check or clear the box for a category to select or unselect all of the Metrics in the category. As you click on an metric in the selection pane, its corresponding description displays in the lower pane of the dialog box. The descriptions include brief explanations of what the metric looks for, examples of violations, and advice on how to correct the code.
Options pane	The set of options vary depending on the selected metric. Where necessary, option controls are explained in the description for the particular metric. Use the toolbar buttons to display properties in the desired manner. Other options are displayed when applicable to the selected metric.
Button	Description
Categorized	Displays the properties of the metric in expandable groups.
Alphabetic	Displays the properies of the metric in alphabetical order.
Properties	Displays property pages, if any.
Aggregation	Aggregation is always present; Aggregation defines how the metric results are handled. You can select the type of aggregation from the drop-down list (sum, average, maximum, and so on). It is important to note that results are aggregated on each level separately. For example, if you have nested namespaces, only the classes that belong to each namespace are aggregated.
Buttons	

Start	Runs the selected set of Metrics.
Cancel	Discards all changes and closes the dialog box.
Help	Opens this Help topic.

See Also

Quality Assurance facilities overview (see page 98)

Running Metrics (see page 276)

Viewing metric results (see page 276)

3.2.15.24 Rename

Refactor ▶ Rename

Opens when you choose the Rename command from the Refactoring menu, or by using the Refactoring | Rename command on the context menu of code-generating class diagram elements. Renaming is applicable to classes, interfaces, enumerators, structures, delegates, methods, properties, events, and fields.

Warning: This feature is available for implementation projects only.

New name:	Use this field to enter a new name for the element.
Refactor Ancestors:	If this option is checked, the selected member is renamed in the current node element and in its parent elements. This option is available for members only.
Rename Overloads:	If this option is checked, all methods with the same name are also renamed. This option is available for members only.
View references before refactoring:	By default, View references before refactoring is checked. If this option checked when you click Rename, the Refactoring window opens allowing you to review the refactoring before committing to it. If this option is cleared when you click Rename, the Refactoring window opens with the rename completed.
Buttons	
Rename:	Opens the Refactoring window.
Cancel:	Discards all changes and closes the dialog box.

See Also

Refactoring overview (see page 98)

Renaming (see page 189)

3.2.15.25 Safe Delete dialog box

Refactor ▶ Safe Delete

You can open the Safe Delete dialog box from the Refactoring main menu, or by using the Refactoring | Safe Delete command on the context menu of applicable class diagram elements. The Safe Delete command is available for all code-generating class diagram elements. It is not available for namespace elements.

Warning: This feature is available for implementation projects only.

Going to safely delete the following elements:	A read-only field displaying the name of the element to delete.
Usages:	A read-only field reporting if there are any usages of the element.
Buttons	
Delete:	Deletes the element and opens the Refactoring window. This button is active only if there are not any usages found for the element.
View usages:	The button is active only if usages are found. Clicking View usages opens the Refactoring window where you can view the usages before you delete the element.
Cancel:	Closes the dialog without deleting the element.

See Also

Refactoring overview (see page 98)

Safely deleting elements (see page 189)

3.2.15.26 Save Audit and Metric Results dialog box

QA Audits (or: Metrics) pane ▶ Save button

This dialog box is invoked from the audit or metrics results report on pressing the Save button.

Note: This feature is available in implementation projects only.

Select View	Choose the scope of the results to export using the **Select View** list box. Audit results display in tabbed pages in the audit results report view. You can group and ungroup the results using the Group by command on the report view context menu. **Note**: Unless the results have been grouped using the Group by command, the **Active Group** option is not enabled in the dialog box. The possible view options are: **All Results**: If the results are grouped, choosing **All Results** prints a report for all groups in the current tabbed page. If the results are not grouped, then all results export for the current tabbed page. **Active Group**: If the results are grouped, you can select a group in the current tabbed page, and the generated report contains the results from the selected group. **Selected Rows**: You can select single or multiple rows in the audit results report view. Choosing **Selected Rows** generates a report for such selections.
Select Format	Choose the output type from the list box. **XML**: Generates an XML-based report. **HTML**: Generates an HTML-based report.
Checkboxes	Activated when generating an HTML report.
Add Description	If this option is checked, the audit descriptions are saved in a separate folder with hyperlinks to the descriptions from the results file.
Launch Browser	If the option is checked, the generated HTML file is opened in the default viewer.
Select the Destination	Specify the fully qualified path to the destination file, or click the **Browse** button.

Buttons	
OK	Applies specified settings, launches the report generation process, and closes the dialog box.
Cancel	Discards all changes and closes the dialog box.

See Also

Quality Assurance Facilities Overview (see page 98)

Running Audits (see page 273)

Viewing Audit Results (see page 274)

3.2.15.27 Search for Usages dialog box

Search ▸ Search for Usages

This dialog box provides a flexible tool to track references to, and overrides of, elements and members in the source-code projects.

Option	Description
Usages of element itself:	Find references to selected element.
Usages of members:	Find references to members of selected element.
Usages of derived classes:	Find references to derived classes (or interfaces, in case of interface).
Usages of implementations:	Find references to implementing classes (members).
Usages of overloads:	Find references to members that overload the selected one.
Include Usings\Imports:	Find references in using\import statements.
Skip self:	Do not show references that are contained inside the selected element.

See Also

Searching source code for usages (see page 210)

3.2.15.28 Select element dialog box

This dialog box displays a tree view of the available contents within your project groupsolution. Expand the project nodes to reveal the nested classes, select the required element, and click OK when ready.

This dialog box belongs to a general group of selection dialogs where you can choose interactions, operations, ancestor classes, instantiated classes for the objects, etc. This dialog opens when you press the chooser button in a field of the Object InspectorProperties Window, or when More is selected from the Choose Class or Choose Method menu nodes.

See Also

Instantiating a classifier (see page 211)

Role binding (see page 222)

3.2.15.29 Selection Manager

This dialog belongs to a general group of selection dialogs where you can select elements from the available contents and add them to a certain destination scope. All Selection Manager dialogs have a similar structure and varying title strings.

Dialog title:	The title of the dialog varies depending on the way it is invoked. It displays the string that corresponds to the invoking object or property.
Model Elements tab or Diagram elements tab:	The pane on the left of the dialog displays the content available in your project. You can use the explorer to navigate to the element and select it for inclusion in the pane of values returned by the dialog to the invoking object.
Existing and/or ready to add:	This pane displays two kinds of data: Values already existing and passed from the invoking object, if any. Values of the selections you have added from the left-hand pane, if any.
Add:	Enabled when an element is selected in the left-hand pane. Adds the selected element to the right-hand pane.
Remove:	Enabled when you select an item in the right-hand pane. Removes the selected item from the pane. All removed values or objects are removed from the invoking property or diagram upon clicking OK.
Remove All:	Enabled when items are present in the right-hand pane. Removes all items from that pane. All removed values or objects are removed from the invoking property or diagram upon clicking OK.

See Also

Creating a Shortcut (see page 208)

Hyperlinking Diagrams (see page 201)

Hiding (and Showing) Model Elements (see page 232)

3.2.15.30 XMI Export dialog box

File ▶ Export Project to XMI

Use this dialog box to export a Together model to an XML file with the model described in XMI.

To open this dialog box, choose File | Export Project to XMI from the main menu while the project root node is selected in the **Model View**. You can also right-click the project root node in the **Model View** and choose Export Project to XMI from the context menu.

Select XMI type	Choose the required XMI type from the list of supported types. Some of the elements supported by Together are not allowed in IBM Rational Rose. The option, UML 1.3 (with Unisys Extension recommended for Rose), drops all Rose-inconsistent elements. If such messages are encountered, appropriate messages appear in the Output tab.
XMI Encoding	Click the drop-down arrow to choose the required encoding of the output stream.
Select the Export Destination:	Specify the fully-qualified name of the output XML file. Use the Browse button to define the target location.
Buttons	
Export	Accepts the input and exports the model to XMI.
Cancel	Cancels your input and closes the dialog box without exporting the model.

See Also

Import and export features overview (see page 100)

3.2.15.31 XMI Import dialog box

File ▶ Import Project from XMI

Use this dialog box to import an XML file with the model described in XMI.

To open this dialog box, choose File | Import Project from XMI from the main menu while the project root node is selected in the **Model View**. You can also right-click the project root node in the **Model View**, and choose Import Project from XMI from the context menu.

Select the Source File	Specify the fully-qualified name of the XML file. Use the Browse button to navigate to the target location.
Buttons	
Import	Imports the XML file.
Cancel	Cancels your input and closes the dialog without importing the model.

See Also

Import and export features overview (see page 100)

3.2.16 Tools

Topics

Name	Description
CodeGuard Configuration (see page 980)	**Tools ▶ CodeGuard Configuration** Use the **CodeGuard Configuration** dialog box to specify how the CodeGuard runtime debugger behaves. **Note:** CodeGuard is available for only C++ projects.
Tools Options (see page 982)	
Configure Tools (see page 1008)	**Tools ▶ Configure Tools** Indicates which programs are available on the Tools menu.
Edit Object Info (see page 1008)	**Tools ▶ Template Libraries ▶ Properties ▶ Edit button** Use this dialog box to edit information about an object in the **Object Repository**.
Edit Tools (see page 1008)	**Tools ▶ Build Tools ▶ Add or Edit button** Use this dialog box to add or change build tool titles and file associations.
Export Visual Studio Project (see page 1009)	**Tools ▶ Export to Visual Studio...** Use this dialog to convert the current project to a Microsoft Visual Studio project.
History Manager (see page 1009)	The **History Manager** lets you see and compare versions of a file, including multiple backup versions, saved local changes, and the buffer of unsaved changes for the active file. If the current file is under version control, all types of revisions are available in the **History Manager**. The **History Manager** is displayed on the **History** tab, which is in the center of the IDE to the right of the **Code** tab. The **History Manager** contains the following tabbed pages:
Object Repository (see page 1010)	**Tools ▶ Template Libraries ▶ Properties** Use this dialog box to edit, move, and remove form and project template libraries.
Template Libraries (see page 1010)	**Tools ▶ Template Libraries** Adds, edits, and removes template libraries from the IDE.

Tools Properties (see page 1011)	**Tools ▶ Configure Tools ▶ Add and Edit button** Use this dialog box to enter or edit the properties for a program listed on the Tools menu.
XML Mapper (see page 1011)	**Tools ▶ XML Mapper** At design-time, defines the mappings between generic XML documents and the data packets that client datasets use. Each mapping describes the correspondences between the nodes of an XML document and the fields in a data packet. You can define mappings from an existing XML schema (or document) to a client dataset that you define, from an existing data packet to a new XML schema you define, or between an existing XML schema and an existing data packet.
Web App Debugger (see page 1013)	**Tools ▶ Web App Debugger** Acts like a Web server on your development machine. If you build your Web server application as a Web App Debugger executable, deployment happens automatically during the build process. To debug your application, start it using **Run ▶ Run**. Next, select **Tools ▶ Web App Debugger**, click the default URL and select your application in the Web browser that appears. Your application will launch in the browser window, and you can use the IDE to set breakpoints and obtain debugging information.

3.2.16.1 CodeGuard Configuration

Tools ▶ CodeGuard Configuration

Use the **CodeGuard Configuration** dialog box to specify how the CodeGuard runtime debugger behaves.

Note: CodeGuard is available for only C++ projects.

Item	Description
Enable (CodeGuard)	Enables or disables CodeGuard.
Stack Fill Frequency	Specifies how frequently CodeGuard fills the uninitialized portion of the runtime stack with a unique byte pattern. Values are: • **-1** = Never • **0** = After every call to a runtime function covered by CodeGuard. • **n [0...15]** = After every 2^n calls to a runtime function covered by CodeGuard. For example, if n is 1, then the stack is filled every other time a runtime function is called.
Statistics	Reports function and resource usage statistics.
Resource Leaks	Reports resource leaks detected after the application terminates.
Send To OutputDebugString	Uses the `OutputDebugString` function to send CodeGuard messages to an external debugger.
Append To Log File	Appends the error log to the existing log. When this option is disabled, CodeGuard writes over the existing error log.
Repeated Errors	Reports errors that occur repeatedly per function.
Limit Number Of Error Messages	Limits the number of errors reported. You can specify a maximum value of 65535.
Enable (Error Message Box)	Enables the **Error Message Box**. If you run a CodeGuard-enabled application outside of RAD Studio, the **Error Message Box** displays when runtime errors occur.
Caption	Specifies the text that appears in the title bar.
Message	Specifies the error message to display.
Read Debug Info	Enables CodeGuard to use the debugging information in your project to point to a source line when a runtime error is reported.

Source Path	If the source code is in a different location from the executable, specify the path (or paths separated by semicolons). CodeGuard checks its own debug source path first, then (if it is running in the IDE) checks the IDE debug source path.

Resource Options

Use the **Resource Options** page to specify how CodeGuard covers various types of resources.

Item	Description
Resources	Lists the resource types that CodeGuard can cover, as follows: **Memory Block** — Memory managed by `malloc` and `free` functions. **Object** — Memory managed by **new** and **delete** operators. **Object Array** — Memory managed by **new[]** and **delete[]** operators. **File handle** — A file managed by the `open` and `close` functions. **File stream** — A file managed by the `fopen` and `fclose` functions. **Pipe stream** — A command processor pipe managed by the `_popen` and `_pclose` functions. **Directory stream** — A directory managed by the `opendir` and `closedir` functions.
Enable Tracking	Enables tracking on the selected resource. Disabling tracking results in lower memory usage and faster execution.
Track Resource Leaks	Reports resource allocations that have no matching deallocations. For example, a leak can be caused by failing to free a file handle before the program terminates.
Report Invalid Handle / Resource Parameters	Reports incorrect usage of resources in function arguments.
Delay Free	Tracks the selected resource after it has been deallocated. When you enable the **Delay Free** option, CodeGuard marks the each resource once it has been freed and prevents Windows and runtime libraries from attempting to reuse the resource. Some resources, such as stack memory allocations, cannot be queued for delayed release.
Delay Queue Length	Specifies the number of objects that can be queued for delayed release. You can set a maximum value of 65535 objects.
Maximum Memory Block Size	Specifies the maximum memory block size that CodeGuard can store in the delay queue. You can set a maximum value of 65535 bytes.

Function Options

Use the **Function Options** page to specify how CodeGuard covers various types of functions.

Item	Description
Functions	Lists the functions that CodeGuard can track.
Disable Function Tracking	Disables function tracking for the selected functions.
Memory Access Errors	Reports a runtime error if a function uses a pointer to reference invalid memory.
Log Each Call	Reports each call to the selected functions.
Warnings	Reports situations where your application may be accessing memory beyond a buffer's maximum size. Warnings are reported for only the following runtime library functions: `strncmp`, `strnicmp`, `strncmpi`, `_fstrncmp`, `_fstrnicmp`, `memcmp`, `memicmp`, `_fmemcmp`, `_fmemicmp`, `fnmerge`, `fnsplit`, `getcurdir`.
Function Results Errors	Reports if the selected functions return a value that indicates failure.

Invalid Handle / Resource Parameters	If any of the selected functions' parameters is a handle or resource identifier, verify that it has been properly allocated and is currently valid.
Set Default Function Options	Displays the **Set Default Function Options** dialog box, which you can use to view and set the default function options.
Reset To Default Function Options	Applies the default function options to the selected functions.

Ignored Modules

Use the **Ignored Modules** page to specify modules that you want CodeGuard to skip when it reports errors.

See Also

CodeGuard overview

Using CodeGuard

3.2.16.2 Tools Options

Topics

Name	Description
ASP.NET (↗ see page 984)	**Tools ▶ Options ▶ HTML/ASP.NET Options ▶ ASP. NET** Use this dialog box to set default information for creating ASP .NET Web applications.
Add Exception Range (↗ see page 985)	**Tools ▶ Options ▶ Debugger Options ▶ Native OS Exceptions ▶ Add button** Use this dialog box to specify the range of exceptions on which you want the product to break execution. The numeric value that you associate with each exception will be displayed at the bottom of the **Exceptions** list on the **Native OS Exceptions** page of the **Debugger Options** dialog box.
Add Language Exception (↗ see page 985)	**Tools ▶ Options ▶ Debugger Options ▶ CodeGear .NET Debugger ▶ Language Exceptions ▶ Add button** Use this dialog box to add a language exception to the list on the **Language Exceptions** page. To add the **Notify on Language Exception** button to a toolbar, use **Tools ▶ Toolbars ▶ Customize**.
Apply Updates (↗ see page 985)	Use this dialog box to review and confirm proposed changes to the sources when you refresh, save, or register the type library by using the Type Library editor.
CodeGear Debuggers (↗ see page 986)	**Tools ▶ Options ▶ Debugger Options ▶ CodeGear Debuggers** Use this page to set the debugger options for the IDE.
Paths and Directories (C++) (↗ see page 987)	**Tools ▶ Options ▶ Environment Options ▶ C++ Options ▶ Paths and Directories** Use the **Paths and Directories** page to specify directories, compiler, and linker options for all packages.
Type Library (C++) (↗ see page 988)	**Tools ▶ Options ▶ Environment Options ▶ C++ Options ▶ Type Library** Use the **Type Library** page to select options for the **Type Library** editor.
Code Insight (↗ see page 988)	**Tools ▶ Options ▶ Editor Options ▶ Code Insight** Use this page to configure how Code Insight works while editing code in the **Code Editor**. **Note:** HTML and CSS support only the Code Completion , **Error Insight**, and **Code Template Completion** features.
Colors (↗ see page 989)	**Tools ▶ Options ▶ Editor Options ▶ Code Insight ▶ Colors** Use the **Colors** page to change the appearance of the **Code Completion** window.
Color (↗ see page 989)	**Tools ▶ Options ▶ Editor Options ▶ Color** Use this page to specify how the different elements of your code appear in the **Code Editor**.
Debugger Options (↗ see page 990)	**Tools ▶ Options ▶ Debugger Options** Use this page to set general debugger options for the IDE.
VCL Designer (↗ see page 990)	**Tools ▶ Options ▶ VCL Designer** Use this page to specify preferences for the VCL Forms Designer.

Library (see page 991)	**Tools ▶ Options ▶ Delphi Options ▶ Library** Use this page to specify directories, compiler, and linker options for all packages.
Display (see page 992)	**Tools ▶ Options ▶ Editor Options ▶ Display** Use this page to set display and font options for the **Code Editor**.
Editor Options (see page 993)	**Tools ▶ Options ▶ Editor Options** Use this page to customize the behavior of the **Code Editor**.
Environment Options (see page 994)	**Tools ▶ Options ▶ Environment Options** Specifies IDE configuration preferences.
Environment Variables (see page 995)	**Tools ▶ Options ▶ Environment Options ▶ Environment Variables** Use this page to view system environment variables and to create, edit, and delete user overrides.
Event Log Options (see page 995)	**Tools ▶ Options ▶ Debugger Options ▶ Event Log** Use this dialog box to control the content, size, and appearance of the event log.
Explorer (see page 996)	**Tools ▶ Options ▶ Environment Options ▶ Explorer** Use this page to control the behavior of the **Structure** view and **Project Manager**. **Note:** Right-click an item in the Structure view and choose Properties to display this page as a separate **Explorer Options** dialog box.
<generic_ordered_list> Dialog Box (see page 997)	This generic dialog box can have several different titles (such as **Conditional Defines**, **Directories**, or **Include path**), and the box is typically invoked from a field on either theTools->Options or Project->Options dialog box. Use this generic dialog box to manage an ordered list of items, such as paths or defines. **Note:** Not all of the options described below are available for all occurrences of this dialog.
HTML/ASP.NET Options (see page 998)	**Tools ▶ Options ▶ HTML/ASP.NET Options** Use this dialog box to specify preferences for editing HTML on the **Designer**.
HTML Formatting (see page 998)	**Tools ▶ Options ▶ HTML/ASP.NET Options ▶ HTML Formatting** Use this dialog box to specify formatting preferences for auto-generated HTML on the **Code** tab.
HTML Tidy Options (see page 999)	**Tools ▶ Options ▶ HTML/ASP.NET Options ▶ HTML Tidy Options** Use this dialog box to control how HTML Tidy formats HTML in the **Code** tab. HTML Tidy is the standard "pretty print" formatting tool from www.w3c.org.
Language Exceptions (see page 999)	**Tools ▶ Options ▶ Debugger Options ▶ Language Exceptions** Use this page to configure how the debugger handles thrown language exceptions. The debugger always stops on unhandled exceptions.
Native OS Exceptions (see page 999)	**Tools ▶ Options ▶ Debugger Options ▶ Native OS Exceptions** Use this dialog box to determine how exceptions are handled by the debugger. Select an exception from the list and adjust the **Handled By** and **On Resume** options.
New Tags (see page 1000)	**Tools ▶ Options ▶ HTML/ASP.NET Options ▶ HTML Tidy Options ▶ New Tags** Use this page to list tags that would normally cause HTML Tidy to issue a warning or error, such as ASP tags.
Override System Variable/New User Variable/Edit User Variable (see page 1000)	**Tools ▶ Options ▶ Environment Options ▶ Environment Variables ▶ Add Override, New, and Edit buttons** or **Project ▶ Options ▶ Debugger ▶ Environment block ▶ Add Override, New, and Edit buttons** Use this dialog box to create or modify user overrides for system variables.
Object Inspector (see page 1000)	**Tools ▶ Options ▶ Environment Options ▶ Object Inspector** Use this page to configure the **Object Inspector**. You can also access this page by right-clicking the **Object Inspector** and choosing **Properties**.
Source Control Options (see page 1001)	**Tools ▶ Options ▶ Source Control Options** Use this page to set source control system options.
Source Options (see page 1001)	**Tools ▶ Options ▶ Editor Options ▶ Source Options** Use this page to configure **Code Editor** settings for various types of source files.
Colors (see page 1002)	**Tools ▶ Options ▶ Environment Options ▶ Tool Palette ▶ Colors** Use this dialog box to change the colors of the **Tool Palette**.
Tool Palette (see page 1003)	**Tools ▶ Options ▶ Environment Options ▶ Tool Palette** Use this dialog box to change the appearance of the **Tool Palette**.

Color (↗ see page 1004)	**Tools ▶ Options ▶ Translation Tools Options ▶ Color** Use this dialog box to define a color scheme for the Translation Manager.
Font (↗ see page 1004)	**Tools ▶ Options ▶ Translation Tools Options ▶ Font** Use this dialog box to set font preferences for the Translation Manager.
Form Designer (↗ see page 1004)	**Tools ▶ Options ▶ Translation Tools Options ▶ Form Designer** Use this dialog box to specify preferences for the forms displayed while using the Translation Manager.
Packages (↗ see page 1005)	**Tools ▶ Options ▶ Packages** Use this dialog box to add or remove designtime packages from the resource project in the External Translation Manager.
Translation Tools Options (↗ see page 1005)	**Tools ▶ Options ▶ Translation Tools Options** Use this dialog box to configure the Satellite Assembly Wizard, Resource DLL Wizard, Translation Manager, and Translation Repository.
Repository (↗ see page 1006)	**Tools ▶ Options ▶ Translation Tools Options ▶ Repository** Use this dialog box to configure the Translation Repository.
Translation Repository (↗ see page 1006)	**View ▶ Translation Manager ▶ Translation Repository** Use the Translation Repository dialog to find, edit, and delete resource strings. While in the Translation Manager, you can use the Translation Repository to store and retrieve translated strings. By default, the Translation Repository stores data in `default.tmx`, located in the RAD Studio`/bin` directory. Use the toolbar icons to create, open, and save a Translation Repository `.tmx` file. After opening a `.tmx` file, you can use the right-click context menu commands to perform actions on individual resource strings. **Tip:** To configure the Translation Repository, close it and choose Tools->Options->Translation Tools Options->Repository .
Type Library (Delphi) (↗ see page 1007)	**Tools ▶ Options ▶ Environment Options ▶ Delphi Options ▶ Type Library** Use this dialog box to select options for the Type Library editor.
WebSnap (↗ see page 1007)	**Tools ▶ Options ▶ WebSnap** Use this page to examine and set WebSnap options.

3.2.16.2.1 ASP.NET

Tools ▶ Options ▶ HTML/ASP.NET Options ▶ ASP. NET

Use this dialog box to set default information for creating ASP .NET Web applications.

Browser options	Description
Name	Indicates the internet browser used to open the Web application when you run it within the IDE.
Path	Indicates the path to the internet browser executable file. To change the path, select **Other** from the **Name** drop-down list and then use the browse button to navigate to a browser executable file.
Parameters	Optional. Enter parameters to be passed to the internet browser application.

Cassini web server options	Description
Path	Indicates the path to the Cassini Web server executable file.
Port	Indicates the TCP/IP port used by the Cassini Web server.
Start Cassini	Use the drop-down menu to choose when the Cassini web server starts: when the project opens or when the project runs.

New web project defaults options	Description
Base Directory	Indicates the default directory path used for new Web applications. This path is displayed on the **New ASP .NET Application** dialog box, however, you can change the path as needed.
Web Server	Indicates the default Web server for new Web applications. When you create a new Web application, this server is displayed on the **New ASP .NET Web Application** dialog box. However, you can change the server as needed.

Note: The Cassini Web Server can be downloaded from http://www.asp.net/Projects/Cassini/Download.

3.2.16.2.2 Add Exception Range

Tools ▸ Options ▸ Debugger Options ▸ Native OS Exceptions ▸ Add button

Use this dialog box to specify the range of exceptions on which you want the product to break execution. The numeric value that you associate with each exception will be displayed at the bottom of the **Exceptions** list on the **Native OS Exceptions** page of the **Debugger Options** dialog box.

Item	Description
Range Low	Specify the low value for the range.
Range High	Specify the high value for the range.

Tip: To stop on a single value, specify the same value for the low and high range.

3.2.16.2.3 Add Language Exception

Tools ▸ Options ▸ Debugger Options ▸ CodeGear .NET Debugger ▸ Language Exceptions ▸ Add button

Use this dialog box to add a language exception to the list on the **Language Exceptions** page. To add the **Notify on Language Exception** button to a toolbar, use **Tools ▸ Toolbars ▸ Customize**.

Item	Description
Text Box	Enter any intrinsic, framework-defined, or user-defined language type, for example, `System.Windows.Forms.Panel`, `Project123.WinForm`, or `MyClass`.

3.2.16.2.4 Apply Updates

Use this dialog box to review and confirm proposed changes to the sources when you refresh, save, or register the type library by using the Type Library editor.

Item	Description
Select Updates	Displays the changes, in order, that will be made to your project. Check or uncheck the box next to each change to include or exclude the changes to that file. If you uncheck a change on which later changes depend (for example the creation of a file to which later changes add code), the later changes are automatically unchecked.

Details	Displays all the changes that will be added to implement the currently selected change. When you click **OK**, the changes in this edit window, including any modifications you make within the dialog, are added for every update checked in the Select Updates list. If an update consists of new code that is added to a file, the **Details** box shows a single edit control that displays the new code. If the update modifies existing code, the Details page shows two text windows: the first is the new code that reflects the modifications, and the second shows the original code that has been changed.
Don't show this dialog again	Prevents this dialog box from being displayed each time you modify a type library and attempt to refresh, save, or register the type library. Check this box to implement changes without checking with you. Checking this box also unchecks the **Display updates before refreshing** option on the **Type Library** page of Tools ▶ Options ▶ Environment Options ▶ Delphi Options.

3.2.16.2.5 CodeGear Debuggers

Tools ▶ Options ▶ Debugger Options ▶ CodeGear Debuggers

Use this page to set the debugger options for the IDE.

Item	Description
Allow side effects in new watches	Causes the watch to be evaluated even if doing so would cause side effects. Can be set for individual watches using the **Watch Properties** dialog box. By default, this option is not set.
Multiple evaluators	Specifies that the appropriate evaluator (C++ or Delphi) is used for each module that is loaded in the process you are debugging. For example, if your Delphi program loads a dll build with the C++ personality, the C++ evaluator is used when debugging into the C++ dll. If you uncheck this option, only the evaluator that is appropriate for the active personality is used. Note that this option is only available for the CodeGear Win32 Debugger.
Debug spawned processes	Debugs processes that are spawned by the process you are debugging. If not checked, spawned processes are run but are not under control of the debugger.
Ignore non-user breakpoints	Breaks only at breakpoints you have explicitly set using the IDE. When this option is checked, the native debugger ignores hardcoded `int 3` breakpoints as well as breakpoints that result from a call to the Windows API method `DebugBreak` . Additionally, pressing F12 while a native application is running does not break into the debugger when this option is checked. The managed debugger also ignores breakpoints that result from a call to `System.Diagnostics.Debugger.Break`. When you change this option, the change takes effect immediately. The default value is Off.
Show inherited	Switches the **Debug Inspector** dialog Data, Methods, and Properties panes between two modes: one that shows all intrinsic and inherited data members or properties of a class, or one that shows only those declared in the class. For class objects, this lets you determine whether you see members that are part of an ancestor class or only members declared in the immediate class whose object you are inspecting.
Show fully qualified names	Shows inherited members using their fully qualified names.
Sort by name	Alphabetically sorts the pages of the **Debug Inspector**. If this option is not selected, the pages are sorted by declaration order. Note that this option is ignored in Delphi for Win32 projects.
Inspectors stay on top	Keeps all debugger inspector windows visible even if they are inactive.
Embedded in editor	Specifies that the Disassembly view comes up as an integral part of the CPU view. This is the default.
Separate dockable window	Specifies that the Disassembly view comes up as a separate window that you can move around in the IDE.
Debug Symbols Search Path	Specifies the path to your debug symbols files (`.pdb`) and `.tds` files. These files are normally stored with your executable, or dynamic link library (DLL).

Debug Source path	Specifies directories where the CodeGear debuggers look for unit files that cannot be found on the project search path or project source path. Additional directories are searched in the following order: 1. Project-specific **Debug Source path**, specified on the **Project▶Options▶Debugger** page. 2. **Browsing path**, specified as follows: • For Delphi for Win32: on **Tools▶Options▶Environment Options▶Delphi Options▶Library — Win32**. • For Delphi.NET, on **Tools▶Options▶Environment Options▶Delphi Options▶Library — NET**. • For C++, on **Tools▶Options▶Environment Options▶C++ Options▶Paths and Directories**. 3. **Debug Source path** (this option), for projects that do not have a project-specific **Debug source path** and for debugging with no project loaded. If no project is loaded in the IDE, only the directories specified with this option are searched.

See Also

Setting the Search Order for Debug Symbol Tables (see page 129)

3.2.16.2.6 Paths and Directories (C++)

Tools▶Options▶Environment Options▶C++ Options▶Paths and Directories

Use the **Paths and Directories** page to specify directories, compiler, and linker options for all packages.

Item	Description
Include Path	Specifies the directories with header files used by default for C++ projects.
Library Path	Specifies where to find C++ header and library files for installed components and packages.
Package output directory	Specifies where the compiler should put compiled packages files.
BPI / LIB output directory	Specifies where package `bpi` and `lib` files are placed by default. The `lib` is a static library and the `bpi` is an import library. The default specified here can be overridden in individual packages by using the package options.
Browsing path	Specifies directories where the **Project Browser** searches for files when it cannot find an identifier on the project search path or source path.
Restrict refactoring path	Specifies the directories that you want to restrict from refactoring. For example, if you attempt to rename a symbol, and a reference is found in a file on one of the paths specified here, the refactoring ends with an error message. Two built-in directories are automatically restricted from refactoring: `$(BDS)\include\vcl` and `$(BDS)\include\dinkumware`. These two paths restrict refactoring from VCL sources and the Dinkumware STL sources. If you are using other third-party libraries (such as Boost), enter in this field the path where the library header files are located.

Tip: You can enter a path relative to the RAD Studio root directory using the $(BDS)

environment variable.

3.2.16.2.7 Type Library (C++)

Tools ▸ Options ▸ Environment Options ▸ C++ Options ▸ Type Library

Use the **Type Library** page to select options for the **Type Library** editor.

Item	Description
Use dispinterfaces in control wrappers	If the component supports both `vtable` and `IDispatch`-based interfaces, checking this option causes the importer to make the `dispinterface` the default interface for the component. The default behavior is to make the `vtable`-based interface the default interface.
MS-style property getter/setter prefixes	If this option is checked, the importer uses Microsoft Visual C++ style prefixes on property getter and setter methods. Otherwise, the default prefixes get_ and set_ are used.
Change suffix	The type library importer appends the suffix `_OCX` to the component wrapper files it generates. You can change this behavior by clicking **Change suffix** and typing a new suffix in the text field.
Ignore special CoClass Flags when importing	When you import an ActiveX Control, the type library importer only imports CoClasses that are not marked as **Hidden**, **Restricted**, or **Predefined**, and marked as **CanCreate**. These flags are supposed to be set if the object is intended for general use. However, if you want to create a control for an internal application only, you can override the flags to generate the CoClass wrappers. In this case, you would check **Ignore special CoClass flags when importing**, **Hidden**, **Restricted**, and uncheck **CanCreate**. Check the coclass flags you want to ignore when importing ActiveX controls.
Predefined	Client applications should automatically create a single instance of this object.
Restricted	A coclass marked **restricted** is supposed to be ignored by tools that access COM objects. It is exposed by the type library but restricted to those authorized to use it.
Hidden	The interface exists but should not be displayed in a user-oriented browser.
Can Create	The instance can be created with CoCreateInstance.

3.2.16.2.8 Code Insight

Tools ▸ Options ▸ Editor Options ▸ Code Insight

Use this page to configure how Code Insight works while editing code in the **Code Editor**.

Note: HTML and CSS support only the Code Completion

, **Error Insight**, and **Code Template Completion** features.

Item	Description
Source file type	Displays a list of programming languages for which you can use Code Insight features. You can specify different Code Insight options for each language.
Use Editor Font	Use the same font as the Code Editor instead of the standard IDE font.
Code completion	Displays a list of properties, methods and events when you enter a class name followed by a period in the Code Editor. You can then select an item and press `ENTER` to add it to your code. If this option is not checked, you can still invoke code completion by pressing `CTRL+SPACE`. The default value is On (checked).
Error Insight	Underlines invalid code and HTML in red. Positioning the cursor over invalid text displays a tooltip window containing the probable cause of the error. The default value is On (checked).

Code Template Completion	Automatically adds a code template when you type a token that starts a template and press TAB. The default value is On (checked).
Auto Complete Templates	Invokes code template completion when you press SPACE after you begin an existing template. When this option is disabled, you must press TAB to invoke template completion after you type in the template name. The default value is On (checked).
Template Hints	Enables template hints. Template hints appear when you add a template in the **Code Editor** and tab between the preset cursor positions in the template. The default value is Off (unchecked).
Delay	Sets the duration of the pause before a **Code Insight** window displays. Select from None, Low, Medium or High.

See Also

Code Editor Overview (↗ see page 42)

Using Code Insight (↗ see page 146)

Using Class Completion (↗ see page 145)

3.2.16.2.9 Colors

Tools ▶ Options ▶ Editor Options ▶ Code Insight ▶ Colors

Use the **Colors** page to change the appearance of the **Code Completion** window.

Item	Description
Source file type	Displays a list of programming languages for which you can use Code Insight features. You can specify different Code Insight options for each language.
Code Completion Listbox Colors	Specifies the color for each component of the **Code Completion** window.

See Also

Code Editor Overview (↗ see page 42)

Using Code Insight (↗ see page 146)

Using Class Completion (↗ see page 145)

3.2.16.2.10 Color

Tools ▶ Options ▶ Editor Options ▶ Color

Use this page to specify how the different elements of your code appear in the **Code Editor**.

Item	Description
Color SpeedSetting	Enables you to quickly configure the **Code Editor** display using predefined color combinations. Select a Color SpeedSetting from the drop-down list and look at the sample code window to see how the settings will appear in the **Code Editor**.
Bold	Applies bold formatting to the selected element.
Italic	Italicizes the selected element.
Underline	Underlines the selected element.

Foreground	Displays the code element using default system colors for the foreground. Unchecking this option restores the previously selected color or, if no color has been previously selected, sets the code element to the system color.
Background	Displays the code element using default system colors for the background. Unchecking this option restores the previously selected color or, if no color has been previously selected, sets the code element to the system color.
Element	Specifies syntax highlighting for a particular code element. You can choose from the **Element** list box or click the element in the sample **Code Editor**. As you change highlighting on code elements, you can see the effect in sample code window.
Foreground Color	Sets the foreground color for the selected code element. The foreground color changes automatically for each element you choose in the **Element** list box.
Background Color	Sets the background color for the selected code element.
Language display pane	Click on the language tab to see how option choices affect appearance of that language's code source.

Note: The foreground color and background colorsof the Modified line

item in the **Element** list are the colors used to mark lines modified since last save and lines modified and saved in the current session, respectively.

3.2.16.2.11 Debugger Options

Tools ▶ Options ▶ Debugger Options

Use this page to set general debugger options for the IDE.

Item	**Description**
Integrated Debugging	Activates the Integrated Debugger.
Map TD32 keystrokes on run	Allows you to use the keystrokes from TD32 (Turbo Debugger 32-bit) in the IDE. When this option is checked, the TD32 keymapping will be active anytime a debug session is in progress. Note that if this option is selected, the **Mark buffers read-only on run** option is automatically checked as well and cannot be unchecked.
Mark buffers read-only on run	Marks all editor files, including project and workgroup files, read-only when the program is run. When this option is selected, it will not change the attributes of the files after the program terminates. If the file was not marked read-only before running the program, the product will change the attributes of the file back to their original configuration after the program terminates.
Rearrange editor local menu on run	Moves the Debugger area of the Code Editor context menu to the top when you run a program from the IDE, for easier access the Debugger commands. Display the **Code Editor** context menu by right-clicking anywhere in the **Code Editor** window.
Automatically close files implicitly opened while debugging	Closes all files that you did not explicitly open. If the debugger opened a file implicitly, that file is closed automatically at the end of the debug session if it has not been changed. Files that have been edited or for which you have set a breakpoint are not closed automatically.
Registered debuggers	An information-only list of the registered debuggers in the system. The current active debugger is displayed in boldface. (Only displayed when there is more than one registered debugger.)

3.2.16.2.12 VCL Designer

Tools ▶ Options ▶ VCL Designer

Use this page to specify preferences for the VCL Forms Designer.

Item	Description
Display grid	Displays a grid of dots on the Designer to aid in aligning controls.
Use designer guidelines	Enables guidelines on the **Forms Designer**. Guidelines facilitate the alignment of components on a form.
Snap to grid	Automatically aligns controls on the Designer to the nearest grid line as you move the control.
Grid size/Snap tolerance	Sets grid spacing in pixels along the x- and y-axis. Specify a higher number increase grid spacing.
Show Component Captions	Displays captions for nonvisual components you drop on a form or data module.
Show Designer hints	Displays a class name in a tooltip for a nonvisual component on a form or data module.
Show extended control hints	Displays a tooltip for controls that include the origin (position on the form), size (width and height), tab stop (whether the user can tab to a control), and order that you added the control to the form. Disabled if **Show Designer hints** is turned off.
Embedded Designer	Displays VCL Forms on the **Design** tab next to the **Code** tab. If unchecked, VCL Forms are displayed as undocked, floating windows, which is useful for viewing both the form and code at the same time. If unchecked, you must use either the **Classic Undocked Layout** or the **Default Layout** with the **Dock Edit Window** option unchecked.
Show virtual screen position	Displays the **virtual screen position** view in the lower-right corner of the **Form Designer**. Use this view to quickly set the on-screen runtime position of the form.
New forms as text	Toggles the format in which form files are saved. The form files in your project can be saved in binary or text format. Text files can be modified more easily by other tools and managed by a version control system. Binary files are backward compatible with earlier versions of the product. (You can override this setting on individual forms by right-clicking and checking or unchecking the Text DFM or Text XFM command.)
Auto create forms & data modules	Toggles whether or not to automatically create forms. When unchecked, forms added to the project after the first one are put into the Available Forms list rather than the Auto Create list. You can change where each form is listed by chooing **Project ▶ Options ▶ Forms**.

3.2.16.2.13 Library

Tools ▶ Options ▶ Delphi Options ▶ Library

Use this page to specify directories, compiler, and linker options for all packages.

Item	Description
Library path	Specifies search paths where compiler can find the source files for the package. The compiler can find only those files listed in the library path. If you try to build your package with a file not on the library path, you will receive a compiler error.
Package output directory	Specifies where the compiler should put compiled packages files.
DCP/DCPIL output directory	Specifies a separate directory to contain the `.dcp` (Win32) or `.dcpil` (.NET) files.

Browsing path	Specifies the directories where the **Project Browser** looks for unit files when it cannot find an identifier on the project search path or source path. For Win32 and .NET Delphi language projects, the directories specified with this option are appended to the debug source path for the project. So the debugger search order for unit files is determined by the following path settings: 1. The project-specific **Debug Source path**, specified on the **Project ▶ Options ▶ Debugger** page. 2. **Browsing path** (this option). 3. The global **Debug Source path**, specified on the **Tools ▶ Options ▶ Debugger Options ▶ CodeGear Debuggers** page. For C# and Visual Basic projects, or if no project is loaded in the IDE, only the **Debug Source path** specified on the **Tools ▶ Options ▶ Debugger Options ▶ CodeGear Debuggers** page is searched for source files.
Namespace prefixes	Specifies the prefixes for dotted namespaces, to allow you to create a shorthand version of the namespace in the `uses` clause in your code. For example, instead of writing `CodeGear.Vcl.DB`, you could specify `CodeGear.Vcl` as your namespace prefix. In the `uses` clause, you could then specify uses `DB;`.
Debug DCU/DCUIL path	To use this option, you must set **Use Debug DCU/DCUILs** on the **Project ▶ Options ▶ Compiler** page. When that option is set and a path is given, the debugger looks for the `.dcu` (Win32) or `.dcuil` (.NET) files in this path before looking in the unit search path.

Tip: To list multiple values in an edit box, separate the values with a semicolon. Alternatively, click the ellipsis button next to each edit box to add multiple values through an appropriate dialog box.

To specify operating system environment variables in an edit box, use the following syntax:

```
$(VariableName)
```

3.2.16.2.14 Display

Tools ▶ Options ▶ Editor Options ▶ Display

Use this page to set display and font options for the **Code Editor**.

Item	Description
BRIEF cursor shapes	Uses BRIEF editor cursor shapes.
Zoom to full screen	Maximizes the **Code Editor** to fill the entire screen. When this option is off, the **Code Editor** does not cover the main window when maximized.
Sort popup pages menu	Sorts alphabetically the list of pages displayed when you right-click a **Code Editor** tab and click **Pages**. If unselected, the pages are sorted in the order that they were created.
Show image on tabs	Displays an icon on each tab in the **Code Editor**.
Visible right margin	Displays a vertical line at the right margin of the **Code Editor**.
Show line numbers	Displays the current line number and every tenth line number in the left margin of the **Code Editor**.
Number all lines	Displays all line numbers in the left margin of the **Code Editor**.
Visible gutter	Displays the gutter on the left edge of the **Code Editor**.
Right margin	Sets the right margin of the **Code Editor**. The default is 80 characters.
Gutter width	Sets the width of the gutter, default is 30.

Editor font	Select a font type from the available screen fonts installed on your system (shown in the list). The **Code Editor** displays and uses only monospaced screen fonts, such as Courier. Sample text is displayed in the **Sample** box.
Size	Select a font size from the predefined font sizes associated with the font you selected in the **Editor font** list box. Sample text is displayed below the **Sample** box.
Sample	Displays a sample of the selected editor font and size.

3.2.16.2.15 Editor Options

Tools ▶Options▶Editor Options

Use this page to customize the behavior of the **Code Editor**.

Item	Description
Insert mode	Inserts text at the cursor without overwriting existing text. If **Insert Mode** is disabled, text at the cursor is overwritten. (Use the `Ins` key to toggle **Insert Mode** in the **Code Editor** without changing this default setting.)
Group undo	Undoes your last editing command as well as any subsequent editing commands of the same type, if you press `ALT+BACKSPACE` or choose **Edit▶Undo**.
Cursor beyond EOF	Positions the cursor beyond the end-of-file character.
Double click line	Highlights the line when you double-click any character in the line. If this option is not selected, only the selected word is highlighted.
Force cut and copy enabled	Enables **Edit▶Cut** and **Edit▶Copy**, even when there is no text selected.
Auto-complete text to find	Enables auto-complete in the find dialog.
Create backup files	Creates a backup file everytime you update and save a file in the IDE. Backup files are stored in the current directory in a hidden directory named `__history` and can be managed from the **History** tab. Use **File backup limit** to specify the number of backup files maintained in the `__history` directory.
Undo after save	Allows you to retrieve changes after a save.
BRIEF regular expressions	Uses BRIEF regular expressions. Regular expressions assist in pattern-matching operations.
Persistent blocks	Keeps marked blocks selected, even when the cursor is moved using the arrow keys, until a new block is selected.
Overwrite blocks	Replaces a marked block of text with whatever is typed next. If **Persistent blocks** is also selected, text that you enter is appended following the currently selected block.
Find text at cursor	Places the text at the cursor into the **Text To Find** list box in the **Find Text** dialog box when you choose **Search▶Find**. When this option is disabled you must type in the search text, unless the **Text To Find** list box is blank, in which case the editor still inserts the text at the cursor.
Preserve line ends	Preserves end-of-line position.

Editor SpeedSetting	Provides a quick way to set the editor options by using preconfigured settings in the drop-down list. **Default** uses key bindings that match CUA mappings (default). **IDE classic** uses key bindings that match Borland Classic editor keystrokes. **BRIEF emulation** uses key bindings that emulate most of the standard BRIEF keystrokes. **Epsilon emulation** uses key bindings that emulate a large part of the Epsilon editor. **Visual Studio emulation** uses key bindings that emulate a large part of the Visual Studio editor. **Visual Basic emulation** uses key bindings that emulate a large part of the Visual Basic editor.
Undo limit	Indicates the number of keystrokes that can be undone. The default value is 32,767. The undo buffer is cleared each time the product generates code.
File backup limit	If **Create backup files** is checked, controls how many backup files are maintained for files updated and saved in the IDE. The default value is 10, but can be set to 1 through 90. Reducing the **File backup limit** does not cause existing backup files to be deleted from the `__history` directory.

See Also

Default Key Mapping (see page 1073)

IDE Classic Key Mapping (see page 1070)

BRIEF Emulation Key Mapping (see page 1069)

Epsilon Emulation Key Mapping (see page 1076)

Visual Studio Key Mapping (see page 1079)

Visual Basic Key Mapping (see page 1078)

3.2.16.2.16 Environment Options

Tools ▶ Options ▶ Environment Options

Specifies IDE configuration preferences.

Item	Description
Editor files	Saves all modified files in the **Code Editor** when you run, compile, build the project, or exit the product.
Project desktop	Saves the arrangement of your desktop when you close a project or exit the product. When you later open the same project, all files opened when the project was last closed are opened again, regardless of whether they are used by the project.
Show compiler progress	Displays the compilation status of your program as it compiles.
Minimize on run	Minimizes the IDE when you run an application by choosing **Run ▶ Run**. When you close the application, the IDE is restored. When you run an application without using the debugger, the IDE remains minimized.
Hide designers on run	Hides Designer windows, such as the **Object Inspector** and **Alignment Palette**, while the application is running. The windows reappear when the application closes.
Show command line	Displays the command used to compile the project in the **Messages** window when you compile a project. In a C# environment, displays the command used to compile the project and the content of the response file. The response file lists the compiler options and source files to be compiled.
Verbosity	Specifies the verbosity level of the build output. Select Quiet, Minimal, Normal, Detailed, or Diagnostics. The build output is written to the **Output** tab of the **Messages** window.

Auto drag docking	For undocked windows, allows you to dock tool windows by dragging the outline of the one window over another window. If this option selected, pressing the `CTRL` key while dragging a window disables the function. If this option is not selected, pressing the `CTRL` key enables it.
Shared Repository	Specifies the path in which the product looks for the shared repository. Click the **Browse** button to search directories.
Default Project	Specifies the path in which the product looks for default project files. Click the **Browse** button to search directories.

3.2.16.2.17 Environment Variables

Tools ▶ Options ▶ Environment Options ▶ Environment Variables

Use this page to view system environment variables and to create, edit, and delete user overrides.

Item	Description
System variables	Lists all environment variables and their values defined at a system level. You cannot delete an existing system variable, but you can override it.
Add Override	Displays the **Override System Variable** dialog box, allowing you to modify an existing system variable to create a new user override. This button is dimmed until you select a variable in the **System variables** list.
User overrides	Lists all defined user overrides and their values. A user override takes precedence over an existing system variable until you delete the user override.
New	Displays the **New User Variable** dialog box allowing you to create new user override to a system variable.
Edit	Displays the **Edit User Variable** dialog box allowing you to change the user override currently selected in the **User overrides** list.
Delete	Deletes the user override currently selected in the **User overrides** list.

3.2.16.2.18 Event Log Options

Tools ▶ Options ▶ Debugger Options ▶ Event Log

Use this dialog box to control the content, size, and appearance of the event log.

Item	Description
Clear log on run	Causes the event log to be purged at the start of each debug session. If this option is checked while debugging multiple processes, the event log view is cleared when the very first process is started. However, any process started while at least one process is already being debugged will not cause the event log view to be cleared.
Unlimited length	Removes the limit on the length of the event log. When this option is unchecked, set the maximum length of the event log in the **Length** field.
Length	Displays the maximum length of the event log. If the **Unlimited length** check box is checked, this option is inactive. For multiple process debugging, length is the total for the event log, not for a process. The default length is 100.
Scroll new events into view	Controls scrolling of the event log. Disable this option to prevent the event log from scrolling new events into view as they occur. (Set by default.)
Display process info with event	Shows the process name and process ID for the process that generated each event.

Breakpoint Messages	Writes a message to the event log each time a breakpoint or First-chance exception is encountered. The message includes the current EIP address of the program being debugged in addition to information about the breakpoint (pass count, condition, source file name, and line number) or exception. (Set by default.)
Process Messages	Writes a message to the event log each time a process loads or terminates, whenever a module is loaded or unloaded by the process. (Set by default.)
Thread Messages	Writes a message to the event log each time a thread is created or destroyed during a debugging session. (Set by default.)
Module Messages	Writes a message to the event log each time a module (executable, shared object, or package) is loaded or unloaded. It includes the name of the module, its base address, and whether it has debug information. (Set by default.)
Output Messages	Writes a message to the event log each time your program or one of its modules calls OutputDebugString. (Set by default.) This setting is used only by the CodeGear Win32 Debugger.
Application Domain Messages	Writes out a message to the event log each time an application domain is created or unloaded. The application domain creation message precede process load messages for the application. The application event unload message follows the process load messages for the application. (Set by default.) This setting is used only by the CodeGear .NET Debugger.
Managed Debug Assistant Messages	Writes out a message to the event log each time a Managed Debug Assistant is triggered. (Set by default.) This setting is used only by the CodeGear .NET Debugger. More information on Managed Debug Assistants can be found on MSDN at http://msdn2.microsoft.com/en-us/library/d21c150d.aspx.
Windows Messages	Writes a message to the event log for each window message that is sent or posted to one of your application's windows. The log entry will have details about the message, including the message name and any relevant data encoded in its parameters. Messages are not immediately written to the log if your process is running and not stopped in the debugger. As soon as you pause the process in the debugger (by encountering a breakpoint or using Run| Pause) the messages will be written to the event log. (Off by default) This setting is used only by the CodeGear Win32 Debugger.
Use Event Log Colors	Associates colors with specific message types so that the message is displayed in that color in the event log.
Foreground	Sets the color for text that appears in the event log.
Background	Sets the color for the background of the event log.

3.2.16.2.19 Explorer

Tools ▶ Options ▶ Environment Options ▶ Explorer

Use this page to control the behavior of the **Structure** view and **Project Manager**.

Note: Right-click an item in the Structure

view and choose Properties to display this page as a separate **Explorer Options** dialog box.

Item	Description
Highlight incomplete class items	Displays incomplete properties and methods in bold in the **Structure** view .(Not applicable for C++ development.)
Show declaration syntax	Displays the syntax and type of methods or properties. By default, only the names of code elements are displayed in the **Structure** view . (Not applicable for C++ development.)

Explorer Sorting: Alphabetical	Lists source elements alphabetically in the **Structure** view .
Explorer Sorting: Source	Lists source elements in the order in which they are declared in the source file.
Class Completion: Finish incomplete properties	If you write a property declaration, completes the remainder of the declaration for reading and writing that property. If unchecked, class completion applies only to methods. (Not applicable for C++ development.)
Explorer categories	Controls how source elements are categorized in the **Structure** view or **Project Manager**. If a category is checked, elements of that type are grouped under a single node in the tree diagram. If a category is unchecked, each element in that category is displayed independently on the diagram's trunk. The folders in bold take precedence when a conflict exists and an element can appear in two folders. For example, a private field would be listed in the private folder if both **Private** and **Field** were checked. If a folder is checked, the glyph to the right of the check box shows whether the folder is expanded. Click there to expand or close a folder in the **Structure** view. The change goes into effect when you click **OK**.

3.2.16.2.20 <generic_ordered_list> Dialog Box

This generic dialog box can have several different titles (such as **Conditional Defines**, **Directories**, or **Include path**), and the box is typically invoked from a field on either theTools->Options or Project->Options dialog box. Use this generic dialog box to manage an ordered list of items, such as paths or defines.

Note: Not all of the options described below are available for all occurrences of this dialog.

Item	Description
<ordered list>	Lists the items that are to be searched, in the order shown. To add items to this list, use the text field below the list.
⬆ or ⬇	Moves the selected item up or down in the ordered list.
<text_field>	Specifies an item to add or replace in the ordered list. You can type an item in this field, or click an item in the list to select it and display it in this text field.
[...] (Ellipsis)	Displays a dialog box allowing you to navigate to and select a folder. The item that you select is displayed in the text field.
Replace	Replaces the selected item with the item in the text field.
Add	Adds the item in the text field to the ordered list of items.
Delete	Removes the selected item from the ordered list.
Delete Invalid Paths	Removes all greyed paths from the ordered list. A path is greyed if it is no longer valid.
Inherit values from configuration “Base”	Check this box if you want values for this list to be inherited from the Base build configuration.
<display_field>	This display box lists the items that are controlled by the checkbox labeled **Inherit values from configuration “Base.”** Items listed here are grayed to indicate that you cannot enter items into this field or select individual items.

3.2.16.2.21 HTML/ASP.NET Options

Tools ▶ Options ▶ HTML/ASP.NET Options

Use this dialog box to specify preferences for editing HTML on the **Designer**.

Item	Description
Show grid	Displays a grid of dots on the Designer to aid in aligning controls.
Snap to grid	Automatically aligns controls on the Designer to the nearest grid line as you move the control.
Grid size	Sets grid spacing in pixels along the x- and y-axis. Specify a higher number increase grid spacing.
Render HTML controls using...	Applies the current Windows theme to HTML controls on the **Designer** page (if you have Windows XP installed and a theme enabled). This is useful for determining the effect of a theme on controls at design time. At run time, the user's theme settings determines how the controls are displayed.
Insert DIV tag...	Inserts a **<DIV>** tag to indicate a division or section in the HTML file when you press `ENTER`. Otherwise, a **<P>** tag is inserted, indicating a paragraph.
Default Page Layout	Sets the positioning of added components. **Flow Layout**: When a component is dropped, it is positioned in a top to bottom, left to right fashion, flowing similarly to text in word processing. **Grid Layout**: Components are positioned using absolute x- and y-coordinates.
Auto show Smart Tasks when...	If true, a Smart Task window displays after a control is dropped on a Web Form.
Highlight Designer element when...	Highlights the HTML control on the **Designer** when the corresponding tag is edited in the tag editor.
Select highlight color	Activated when **Highlight Designer element when..** is checked. Clicking this button displays a color picker for the HTML control highlight color. Default is yellow.
Web Forms	Used only for ASP.NET. Indicates what is displayed in the edit window when creating a Web document. **Designer**: Displays a WYSIWIG designer. **Markup Editor**: Displays a code editor for markup, such as HTML or ASP code. **Code Editor**: Displays a code editor for programming languages, such as C#, C++, or Delphi.
HTML	Similar to **Web Forms**. Indicates what is displayed in the edit window when creating a Web document. **Designer**: Displays a WYSIWIG designer. **Markup Editor**: Displays a code editor for markup, such as HTML or ASP code.

3.2.16.2.22 HTML Formatting

Tools ▶ Options ▶ HTML/ASP.NET Options ▶ HTML Formatting

Use this dialog box to specify formatting preferences for auto-generated HTML on the **Code** tab.

Item	Description
Using Spaces	Indents generated HTML by using spaces.
Using Tabs	Indents generated HTML by using the tab character.
Size	Specifies the number of spaces used for tab indentation.
Place end tags on the same line	Places closing HTML tags on the same line as the opening HTML tag.
Tags	Indicates whether HTML tags are generated in uppercase or lowercase.
Attributes	Indicates whether HTML tag attributes are generated in uppercase or lowercase.

3.2.16.2.23 HTML Tidy Options

Tools ▶ Options ▶ HTML/ASP.NET Options ▶ HTML Tidy Options

Use this dialog box to control how HTML Tidy formats HTML in the **Code** tab. HTML Tidy is the standard "pretty print" formatting tool from www.w3c.org.

Item	Description
HTML Tidy Option Window	List of HTML Tidy options to be used for all HTML formatting within the IDE. You can change each option with a context menu.
Description	Displays a description of the selected HTML Tidy option.

3.2.16.2.24 Language Exceptions

Tools ▶ Options ▶ Debugger Options ▶ Language Exceptions

Use this page to configure how the debugger handles thrown language exceptions. The debugger always stops on unhandled exceptions.

Item	Description
Exception Types to Ignore	Lists the types of exceptions you want the debugger to ignore (checked) or not (unchecked). The debugger will not halt execution of your program if the exception raised is listed and checked, or derived from any exception that is listed and checked.
Add	Displays the **Add Exception** dialog box, allowing you to add a user-defined exception to the list.
Remove	Removes the highlighted, user-defined exception from the list. You can not removed default language exceptions from the list.
Notify on Language Exceptions	Halts the execution of your program when your program raises a language exception. If this box is checked, the debugger ignores the exception types you select in **Exception Types to Ignore**. To place this command on your toolbar for easy access, use the **View ▶ Toolbars ▶ Customize ▶ Commands** page.

3.2.16.2.25 Native OS Exceptions

Tools ▶ Options ▶ Debugger Options ▶ Native OS Exceptions

Use this dialog box to determine how exceptions are handled by the debugger. Select an exception from the list and adjust the **Handled By** and **On Resume** options.

Item	Description
Exceptions	Lists the native operating system exceptions and any user-defined exceptions.
Handled By	Specifies whether the exception will be handled by the debugger or by your program. If you have added exception handling to your project, select **User Program**.
On Resume	Specifies whether the product will continue to handle the exception, or whether the project will run unhandled.
Add	Displays the **Add Exception Range** dialog box, allowing you to add user-defined exceptions to be handled by the debugger.
Remove	Removes the selected user-defined exception from the list. Native operating system exceptions can not be removed.

3.2.16.2.26 New Tags

Tools ▶ Options ▶ HTML/ASP.NET Options ▶ HTML Tidy Options ▶ New Tags

Use this page to list tags that would normally cause HTML Tidy to issue a warning or error, such as ASP tags.

Item	Description
Block level tags	Enter the tags that HTML Tidy should process as block tags. Omit the begin (<) and end (>) symbols. Separate multiple tags by a comma, for example: `asp:button,asp:checkbox`
Empty tags	Enter the tags that HTML Tidy should process as empty inline tags. Omit the begin (<) and end (>) symbols. Separate multiple tags by a comma.
Inline tags	Enter the tags that HTML Tidy should process as non-empty inline tags. Omit the begin (<) and end (>) symbols. Separate multiple tags by a comma.
Pre tags	Enter the tags that HTML Tidy should process the same way it processes the HTML <PRE> tag. Omit the begin (<) and end (>) symbols. Separate multiple tags by a comma.

3.2.16.2.27 Override System Variable/New User Variable/Edit User Variable

Tools ▶ Options ▶ Environment Options ▶ Environment Variables ▶ Add Override, New, and Edit buttons

or

Project ▶ Options ▶ Debugger ▶ Environment block ▶ Add Override, New, and Edit buttons

Use this dialog box to create or modify user overrides for system variables.

Item	Description
Variable Name	Type a new variable name or modify an existing one.
Variable Value	Type a new value or modify an existing one.

3.2.16.2.28 Object Inspector

Tools ▶ Options ▶ Environment Options ▶ Object Inspector

Use this page to configure the **Object Inspector**. You can also access this page by right-clicking the **Object Inspector** and choosing **Properties**.

Item	Description
SpeedSettings	Displays a drop-down list box to choose from the following color schemes: Custom colors and settings, Default colors and settings, Traditional colors and settings, Classic colors and settings, and Visual Studio(TM) emulation.
Show instance list	Displays the drop-down list box of components and their class names (called the instance list) at the top of the **Object Inspector**. The list is useful when you have many components on your form or data module and can't find the one you want right away.
Show classname in instance list	Displays the component's class name for every component in the instance list, not just the first one.
Show status bar	Displays the status bar at the bottom of the **Object Inspector**. The status bar indicates how many properties or events are not shown as a result of right-clicking the **Object Inspector** and selecting **View**. If all properties or events are visible in the **Object Inspector**, it says **All shown**.

Render background grid	Adds horizontal background lines to designate columns and rows on the **Properties** and **Events** pages.
Integral height (when not docked)	Adjusts the **Object Inspector** between a full row instead of a partial row as you vertically resize the **Object Inspector** with your cursor.
Show read only properties	Displays the properties for components even if the properties are read only. By default, they are grayed out.
Bold non default values	Displays non-default values as bold text in the current **Non Default Value** color setting.
Show gutter	Draws an outline along the left edge of the **Object Inspector** and fills the outlined area with the current **Gutter Color** setting for additional readability.
Colors	To customize one of the imported color schemes, select it from the **SpeedSettings** list. Then select an option and select a different color from the drop-down list below. For example, to change the color of **Value**, the text color for properties' values, select **Value** and click **clYellow** from the **Options** list. You save your new settings once you click **OK**. This automatically saves the changes to the **Custom** colors and settings scheme, not the original scheme. To return to your default settings, click **Default** colors and settings or one of the others.
Expand inline	Displays the properties of the referenced component. To view these properties, click the plus sign (+) next to the referenced component. By default, referenced components are red and their properties green.
Show on events page	Displays the events of the referenced component. By default, referenced properties are red and their events green.

3.2.16.2.29 Source Control Options

Tools ▶Options▶Source Control Options

Use this page to set source control system options.

Item	Description
Source Code Control Providers	Activates a particular source code system provider. The providers must implement the SCC API or a wrapped client that exposes the API.
User Name	Specifies a valid source control system user name. This user name must be defined in the source control system.

3.2.16.2.30 Source Options

Tools▶Options▶Editor Options▶Source Options

Use this page to configure **Code Editor** settings for various types of source files.

Item	Description
Source file type	Choose a predefined or customized source file type.
New	Displays the **New source file type** dialog box, allowing you to create a new file type. Enter a name and click **OK**, and then enter an extension in the **Extensions** drop-down list. You must add an extension if you add a new source file type, or cancel the operation.
Delete	Deletes the predefined or customized file type displayed in the **Source file type** drop-down list box.
Auto indent mode	Positions the cursor under the first nonblank character of the preceding nonblank line when you press ENTER in the **Code Editor**.

Use tab character	Inserts tab characters when you press TAB in the **Code Editor**. If not checked, pressing TAB inserts spaces. If **Smart tab** is enabled, this option is off. To view tab characters, select **Show tab character**.
Smart tab	Tabs to the first non-whitespace character in the preceding line. If **Use tab character** is enabled, this option is off.
Cursor through tabs	Enables the `arrow` keys to move the cursor to the logical spaces within each tab character.
Optimal fill	Begins every auto-indented line with the minimum number of characters possible, using tabs and spaces as necessary.
Backspace unindents	Aligns the insertion point to the previous indentation level (outdents it) when you press BACKSPACE, if the cursor is on the first nonblank character of a line.
Keep trailing blanks	Prevents trailing blanks from being truncated.
Show tab character	Displays tab characters as >>, if **Use tab characters** is selected.
Show space character	Displays typed spaces as dots (.).
Use syntax highlight	Enables syntax highlighting. To set highlighting options, use the **Color** page.
Show line breaks	Displays line break symbols at the end of each line.
Highlight current line	Highlights the current line in the **Code Editor**.
Syntax Highlighter	Choose an option to change the format for displaying code elements. Check **Use syntax highlighting** to enable this option.
Block indent	Specifies the number of spaces to indent a marked block. The default is 2; the upper limit is 16.
Tab stops	Set tabs stops that the cursor will move to when you press TAB. Enter one or more integers separated by spaces. If multiple tab stops are specified, the numbers indicate the columns in which the tab stops are placed. Each successive tab stop must be larger than the previous tab stop. If a single tab stop is specified, it indicates the number of spaces to jump each time you tab.

3.2.16.2.31 Colors

Tools ▶ Options ▶ Environment Options ▶ Tool Palette ▶ Colors

Use this dialog box to change the colors of the **Tool Palette**.

Item	Description
Color Schemes	Lists predefined color combinations. Select a color scheme from the drop-down list to display it immediately in the **Tool Palette**. You can not modify the default color schemes, however, you can select a color scheme and change any of the colors associated with it to create your own unnamed color scheme.
Base Color (Category Colors)	Specifies the color used for the category window background.
Gradient Color (Category Colors)	Specifies the color used for shading the **Base Color**.
Text Color	Specifies the color used for the category captions.
Bold Captions	Applies bold formatting to the category captions.
Vertical Gradient	Applies the **Gradient Color** value to the top of the category window, rather than the left side of the window.
Caption Only Border	Applies the **Base Color**, **Gradient Color**, and **Text Color** values only to the category captions, not the entire category window.
Use +/– Icons	Displays plus (+) and minus (–) signs next to category captions, instead of carets.

Vertical Captions	Displays the category captions vertically on the left side of the category windows.
Normal Color	Specifies the color used for the button background.
Selected Color	Specifies the color used for the background of a button when it is selected and highlighted.
Hot Color	Specifies the color used for the button when you hover the mouse over the button.
Base Color (Background Colors)	Specifies the color used for the frame around each category window.
Gradient Color (Background Colors)	Specifies the color used to shade the frame around each category window.
Gradient Direction	Specifies whether the **Gradient Color** shading around the category window is vertical or horizontal.

Tip: As you change options in this dialog box, the Tool Palette

is automatically updated to show the result of the changes.

See Also

Adding Components to the Tool Palette (see page 160),

3.2.16.2.32 Tool Palette

Tools ▶Options▶Environment Options▶Tool Palette

Use this dialog box to change the appearance of the **Tool Palette**.

Item	Description
Button Size	Changes the size of the icons that represent items on the **Tool Palette**.
Auto Collapse Categories	Allows only one category to be expanded at a time.
Vertical Category Captions	Displays the category captions vertically to the left of items on the **Tool Palette**.
Lock Palette Ordering	Disables drag-and-drop reordering of items on the **Tool Palette**.
Show Button Captions	Displays captions along with item icons.
Vertical Flow Layout	Displays the categories vertically.
Show Palette Wizards	Displays items from the **New Items** dialog box in the **Tool Palette** when the **Code Editor** is active or the **Project Manager** has focus. The **New Items** dialog box is also available by choosing **File▶New▶Other**.
Always Show Designer Items	Displays Designer items, even when the **Code Editor** is active (similar to Delphi 7 behavior). Uncheck this option to omit Designer items from the **Tool Palette** when the **Code Editor** is active.

Tip: As you change options in this dialog box, the Tool Palette

is automatically updated to show the result of the changes.

See Also

Adding Components to the Tool Palette (see page 160)

Adding Components to a Form (see page 152)

Using Code Snippets (see page 148)

3.2.16.2.33 Color

Tools ▶Options▶Translation Tools Options▶Color

Use this dialog box to define a color scheme for the Translation Manager.

Item	Description
Color Scheme	Lets you choose from a selection of predefined color schemes.
Save As	Lets you add your own color scheme to the list.
Remove	Removes a user-defined color scheme from the list.
Element	Lists the elements in the Translation Manager. To set a color for an element, click the element and then click a color in the color box.
Sample	Shows how some of the selected colors will look in the Translation Manager.
User colors	Controls color-coding in the Translation Manager.

See Also

Localizing Applications (↗ see page 18)

3.2.16.2.34 Font

Tools ▶Options▶Translation Tools Options▶Font

Use this dialog box to set font preferences for the Translation Manager.

Item	Description
Grid fonts	Displays the languages and the fonts available to your project.
Sample	Shows a sample of the selected font.
Font	Displays the **Font** dialog box, allowing you to choose a font, font style, font size, effect, color, and script for the language(s) you have selected in the **Grid fonts** box.

See Also

Localizing Applications (↗ see page 18)

3.2.16.2.35 Form Designer

Tools ▶Options▶Translation Tools Options▶Form Designer

Use this dialog box to specify preferences for the forms displayed while using the Translation Manager.

Item	Description
Display grid	Displays dots on the form to make the grid visible.
Snap to grid	Automatically aligns components on the form with the nearest gridline. You cannot place a component in between gridlines.
Grid size	Sets the grid spacing in pixels along the x- and y-axis. Specify a higher number increase grid spacing.

3.2.16.2.36 Packages

Tools ▶Options▶Packages

Use this dialog box to add or remove designtime packages from the resource project in the External Translation Manager.

Item	Description
Add	Lets you navigate to the designtime packages that you want to add to the project. You can add one or more designtime packages at a time.
Remove	Removes the selected designtime packages from the project.

3.2.16.2.37 Translation Tools Options

Tools ▶Options▶Translation Tools Options

Use this dialog box to configure the Satellite Assembly Wizard, Resource DLL Wizard, Translation Manager, and Translation Repository.

Item	Description
Automatically query repository	Automatically populates resource modules with translations for any strings that have matches in the Translation Repository, each time your assemblies are updated. If only one match for a string is found in the Translation Repository, that translated string is copied to the resource modules project. If more than one match is found, the first matching translation in the Repository is copied to the assembly. You can change this behavior by choosing **Tools▶Translation Tools Options**, clicking the **Repository** tab, then changing the **Multiple Find Action** setting.
Automatically compile projects	Compiles projects, without asking first, whenever required by the translation tools (for example, when running the Satellite Assembly Wizard).
Show Translation Manager after wizard	Opens the Translation Manager automatically after running the Satellite Assembly Wizard or Resource DLL Wizard unless one of the resource module projects is active in the Project Manager when you use the wizard.
Automatically quote strings	Supplies required quotation marks around translated strings, unless the strings already contain apostrophes, quotation marks, or control characters (such as #13).
Use 'Newly Translated' Status	When a string is manually translated, or automatically translated from the Translation Repository, the status of the string is changed **Newly Translated** instead of **Translated**. This status is used to determine items which are translated in current translation.
Automatically save files	Saves the current project, without asking first, whenever appropriate (for example, before closing the Translation Manager or running the Satellite Assembly Wizard or Resource DLL Wizard).
Hide empty items	Hides files that do not contain translation items, such as `.nfn` and `.resN` files, in the **Workspace** tab tree view of the Translation Manager. This can improve performance when processing many files, some of which are empty, with the Add strings to repository or Get strings from repository commands.
External editor	Indicates the name of the editor to use when using an external editor in the Translation Manager, for example, `notepad.exe`.

See Also

Localizing Applications (see page 18)

3.2.16.2.38 Repository

Tools ▶Options▶Translation Tools Options▶Repository

Use this dialog box to configure the Translation Repository.

Item	Description
Filename	Sets the location of the Translation Repository, a database for translations that can be shared by different projects. Enter the full name and directory path of the `.tmx` file where the Translation Repository is stored. The default is `$(ETM)\default.tmx`.
Duplicate action	Determines how the repository responds when it finds a duplicate translation string for the same source string. **Skip** does not add the string. **Add** adds the string to the repository if no translated string exists for the original string. **Force Add** always adds the string to the repository, regardless of whether it exists in the repository. **Replace** overwrites the existing string with new string. **Display selection** offers the user a choice.
Multiple find action	Determines how the repository responds when it finds more than one translation for the same source string. **Skip** does not retrieve anything if the repository contains more than one match. **Use first** retrieves the first match. **Display selection** offers the user a choice.

See Also

Localizing Applications (↗ see page 18)

3.2.16.2.39 Translation Repository

View ▶Translation Manager▶Translation Repository

Use the Translation Repository dialog to find, edit, and delete resource strings. While in the Translation Manager, you can use the Translation Repository to store and retrieve translated strings. By default, the Translation Repository stores data in `default.tmx`, located in the RAD Studio`/bin` directory.

Use the toolbar icons to create, open, and save a Translation Repository `.tmx` file. After opening a `.tmx` file, you can use the right-click context menu commands to perform actions on individual resource strings.

Tip: To configure the Translation Repository, close it and choose Tools->Options->Translation Tools Options->Repository

.

See Also

Localizing Applications (↗ see page 18)

Adding Languages to a Project (↗ see page 169)

Editing Resource Files in the Translation Manager (↗ see page 170)

Setting Up the External Translation Manager (↗ see page 172)

3.2.16.2.40 Type Library (Delphi)

Tools ▶ Options ▶ Environment Options ▶ Delphi Options ▶ Type Library

Use this dialog box to select options for the Type Library editor.

Item	Description
SafeCall function mapping	Determines which functions are declared as safecall when declarations specified in Delphi are converted to Interface Definition Language (IDL) in the generated type library. Safecall functions automatically implement COM conventions for errors and exception handling, converting HRESULT error codes into exceptions. If you are entering function declarations in IDL, you must explicitly specify the calling convention as safecall or stdcall. **All v-table interfaces** uses SafeCall for all interfaces. **Only dual interfaces** uses SafeCall only for dual interfaces. **Do not map** does not use the SafeCall calling convention.
Pascal	Delphi language.
IDL	Microsoft Interface Definition Language.
Ignore special CoClass Flags when importing	When you import an ActiveX Control, the type library importer only imports CoClasses that are not marked as **Hidden**, **Restricted**, or **Predefined**, and marked as **Can Create** (actually noncreatable). These flags are supposed to be set if the object is intended for general use. However, if you want to create a control for an internal application only, you can override the flags to generate the CoClass wrappers. In this case, you would check **Ignore special CoClass** flags when importing, **Hidden**, **Restricted**, and uncheck **Can Create** (noncreatable).
Predefined	Client applications should automatically create a single instance of this object.
Restricted	A coclass marked **Restricted** is ignored by tools that access COM objects. It is exposed by the type library but restricted to those authorized to use it.
Hidden	The interface exists but should not be displayed in a user-oriented browser.
Can Create	The instance can be created with CoCreateInstance.
Display updates before refreshing	Displays the **Apply Updates** dialog box, which provides a chance to preview proposed changes to the sources when you try to refresh, save, or register the type library. If this option is not checked, the type library editor automatically updates the sources of the associated object when you make changes in the editor.

3.2.16.2.41 WebSnap

Tools ▶ Options ▶ WebSnap

Use this page to examine and set WebSnap options.

Item	Description
Enable Debugging	Enables the active script debugger when an error occurs while debugging a web page module.
HTML File Extension	Specifies which file extension you want the **New WebSnap Application** wizard to apply to HTML files it generates.
Sample Image File	Used by adapter components to display a sample image in the event that the correct image is not available at design time. Click **Browse** to locate the path for the sample image.

Note: WebSnap is being deprecated in RAD Studio. Although WebSnap is still documented in the online help, the WebSnap product is no longer fully supported. As an alternative, you should begin using IntraWeb (VCL for the Web). IntraWeb is documented in this online help. For more documentation on VCL for the Web, go to http://www.atozed.com/intraweb/docs/.

3.2.16.3 Configure Tools

Tools ▶ Configure Tools

Indicates which programs are available on the Tools menu.

Item	Description
Tools	Lists programs that have been added to the Tools menu.
Add	Displays the **Tools Properties** dialog box, allowing you to add a program.
Delete	Deletes the tool selected in the **Tools** list, allowing you to change the properties of the program.
Edit	Displays the **Tools Properties** dialog box.
Up and Down arrows	Moves the tool selected in the **Tools** list up or down, which changes the order in which the tools appear on the Tools menu.

Tip: Added programs appear at the bottom of the Tools menu.

3.2.16.4 Edit Object Info

Tools ▶ Template Libraries ▶ Properties ▶ Edit button

Use this dialog box to edit information about an object in the **Object Repository**.

Item	Description
Category	Displays the categories that appear in the **New Items** dialog box displayed when you choose **File ▶ New ▶ Other**.
Title	Indicates the title of the selected item.
Description	Indicates the description of the selected item. The description is displayed when you right-click the **New Items** dialog box and choose View Details.
Author	Indicates the name of the author of the selected item.
Browse button	Displays the **Select icon** dialog box, allowing you to select a different icon to represent the object in the **New Items** dialog box. You can use a bitmap of any size, but it will be cropped to 60 x 40 pixels.
New Category	Displays the **New Category Name** dialog box where you enter the name for a new **Object Repository** category.

3.2.16.5 Edit Tools

Tools ▶ Build Tools ▶ Add or Edit button

Use this dialog box to add or change build tool titles and file associations.

Item	Description
Title	Enter the name you want to use for the tool.

Default Extensions	List the file extensions that the tool will compile by default. The tool will be run automatically for files in your project with this extension whenever you compile or build the project. Separate multiple extensions with semicolons. In the **Project Manager**, you can right-click on files with this extension to access the tool.
Other Extensions	List the file extensions for which the tool will be used occasionally. This is useful for tools such as preprocessors, archiving tools, diagnostics, or auto-help generators. Separate multiple extensions with semicolons. In the **Project Manager**, you can right-click on files with this extension to access the tool.
Target Extension	Specify the extension of files that the tool creates (if any).
Command Line	Specify the command line to be executed when a file of this type is built.
Filter	Indicate a custom filter in a package (created using the Tools API) used to filter the tool output information and display it in the **Messages** window. If no filter is specified, a default filter is used. The default filter sends anything written by the tool to stdout and stderr to the **Messages** window.
Macros	Displays a list of macros that you can use in the command line (for example, $NAME gets a file name). The macros are expanded when the tool runs. Double-click the macro or click **Insert** to add it to **Command Line**.
Inserts	Adds the selected macro to **Command Line**.

3.2.16.6 Export Visual Studio Project

Tools ▶ Export to Visual Studio...

Use this dialog to convert the current project to a Microsoft Visual Studio project.

Item	Description
Name	Prefilled with the current project name. You can change the name.

3.2.16.7 History Manager

The **History Manager** lets you see and compare versions of a file, including multiple backup versions, saved local changes, and the buffer of unsaved changes for the active file. If the current file is under version control, all types of revisions are available in the **History Manager**.

The **History Manager** is displayed on the **History** tab, which is in the center of the IDE to the right of the **Code** tab. The **History Manager** contains the following tabbed pages:

Page	Description
Contents	Displays the current and previous versions of the file.
Info	Displays all labels and comments for the active file.
Diff	Displays the differences between the selected versions of the file.

History Manager Toolbar Buttons

Button	Description
⟳	**Refresh revision info** updates the revision list to include unsaved changes to the file.

	Revert to previous revision makes the selected version the current version and is available on the **Contents** and **Info** pages. Reverting a prior version deletes any unsaved changes in the editor buffer.
	Synchronize scrolling synchronizes scrolling in the **Contents** and **Diff** pages and the **Code Editor**. It matches the line of text that contains the cursor with the nearest matching line of text in the other view. If there is no matching text in that region of the file, it matches line numbers.
	Go to next difference repositions the source on the **Diff** page to the next block of different code.
	Go to previous difference repositions the source on the **Diff** page to the previous block of different code.
	Follow text movement locates the same line in the source viewer when switching between views.

The following icons are used to represent file versions in the revision lists.

Revision Icons Used in the History Manager

Icon	Description
🗎	The latest saved file version.
🟢	A backup file version.
👁	The file version that is in the buffer and includes unsaved changes.
🔵	A file version that is stored in a version control repository.
●	A file version that you have checked out from a version control respository.

See Also

IDE Tour (see page 34)

Using the History Manager (see page 149)

3.2.16.8 Object Repository

Tools ▸ Template Libraries ▸ Properties

Use this dialog box to edit, move, and remove form and project template libraries.

Item	Description
Categories	Lists the categories available that contain project and form templates in the **Object Repository**.
Repository Objects	Lists the project and form templates within each category.
Edit	Displays the **Edit Object Info** dialog box, allowing you to edit the properties of templates in the **Object Repository**.
Delete	Removes a template from the **Object Repository**.

3.2.16.9 Template Libraries

Tools ▸ Template Libraries

Adds, edits, and removes template libraries from the IDE.

Item	Description
Name column	Displays the names of each template library listed.
Version column	Displays the current version of each template library listed.
Description column	Displays the descriptions of each template library listed.
Path column	Displays the complete path to the location of each template library.
Properties	Opens a window which displays the Categories in which the template library is listed in the **Object Repository** and the name and description of each object in the template library.
Add	Opens a browser enabling you to select an existing template library to add to the list.
Remove	Deletes the selected template library from the list. It does not delete the template library from the disc.

3.2.16.10 Tools Properties

Tools ▶Configure Tools▶Add and Edit button

Use this dialog box to enter or edit the properties for a program listed on the Tools menu.

Item	Description
Title	Enter a name for the program you are adding. This name will appear on the Tools menu. To add an accelerator to the menu command, precede that letter with an ampersand (&). If you specify a duplicate accelerator, the **Tool Options** dialog box displays a red asterisk (*) next to the program names.
Program	Enter the location of the program you are adding. Include the full path to the program. To search your drives and directories to locate the path and file name for the program, click the **Browse** button.
Working Dir	Specify the working directory for the program. The product specifies a default working directory when you select the program name in the **Program** text box. You can change the directory path if needed.
Parameters	Enter parameters to pass to the program at startup. For example, you might want to pass a file name when the program launches. Type the parameters or use the **Macros** button to supply startup parameters. You can specify multiple parameters and macros.
Macros	Expands the **Tool Properties** dialog box to display a list of available macros. You can use these macros to supply startup parameters for your application. Select a macro and click **Insert** to add the macro to the **Parameters** text box.
Browse	Displays the **Select Transfer Item** dialog box opens, allowing you to navigate to a program.

3.2.16.11 XML Mapper

Tools ▶XML Mapper

At design-time, defines the mappings between generic XML documents and the data packets that client datasets use. Each mapping describes the correspondences between the nodes of an XML document and the fields in a data packet.

You can define mappings from an existing XML schema (or document) to a client dataset that you define, from an existing data packet to a new XML schema you define, or between an existing XML schema and an existing data packet.

Document View Page

This page shows the contents of the currently loaded XML document, represented as a hierarchical tree view. Each node in the tree represents a tag or tag attribute in the XML document. Next to each node is an icon that indicates the type of tag it represents:

Item	Description
⟨E⟩	Represents an element node that is a tag that acts as a parent to other nodes (tags), but does not have a value. The name of the node is the tag name. Typically, element nodes map to datasets (the data packet itself or a nested detail set), although they can also map to fields whose values are a composite of the child node values.
T	Represents a text node. Text nodes represent tagged elements with text values. In the tree, they have the form Nodename="TextValue", where Nodename is the tag name and TextValue is the text that appears between the starting tag and the ending tag. Typically, text nodes map to fields in the corresponding data packet.
A	Represents an attribute node. Attribute nodes correspond to attributes of the parent element's tag in the XML document. In the tree, attributes have the form Nodename="AttributeValue", where Nodename is the name of the attribute and AttributeValue is its value. Typically, A nodes map to fields in the corresponding data packet, where the element for which they are attributes maps to a record.
▤	Represents a nested node. Nested nodes are element nodes that can appear replicated sequentially in the XML document. Typically, nested nodes map to records in the corresponding data packet.
Data View	If unchecked, the hierarchy displays only the names and types of nodes. No values are shown for text or attribute nodes, and only a single instance is shown for any nested nodes. If checked, the hierarchy displays sample values on text and attribute nodes, and repetitions of nested nodes. If you loaded a sample XML file, the **Data View** shows the values stored in that file. If the document was generated from a schema or data packet, sample values are generated for the nodes. When examining a large XML document, it is sometimes easier to uncheck **Data View**, so you can see more of the logical structure with the detailed information removed.

Schema View Page

This page shows the XML schema information. This page has three tabs, which represent the different schema formats supported by XML mapper. These include DTD, XDR (reduced XMLData), and XSD (XML schema). The information on the **Schema View** page can be read from a file or deduced from an actual XML document.

Node Properties Page

This page lets you assign properties to the currently selected node in the XML document pane. These properties are used when generating a transformation file to ensure that data packets generated from XML documents have the correct field types and constraints, and that XML documents generated from data packets have the correct nodes. When you generate a transformation file, it reflects the values currently specified on the **Node Properties** page.

Item	Description
UTF-8 encoded	Controls whether extended characters are encoded using UTF-8 (when checked) or using an escape sequence in the XML (when unchecked). When checked, the Data Format property for Strings, Memos, and WideStrings changes from ANSI to UTF-8.
User Defined Translation	Controls whether the selected node should be transformed automatically. This allows you to perform conversions that are not simple one-to-one mappings that can be specified by giving a data type. For example, you can create a user-defined node to convert an element node that has children for first name and last name into a single "full name" field in the data packet. When you check the **User Defined** check box, you must assign an ID string to represent the node. This ID string is passed to the OnTranslate event handler of TXMLTransform so that you can perform the translation in code. If you do not identify a node as user-defined, the OnTranslate event does not occur for the node.
Node Description	Optional. Enter a description of the node. This description is not added to the XML document or the data packet, but is useful for identifying the purpose of a base set element when you are saving property sets to a node repository file

Tip: To save the current node settings to a node repository file, right-click and choose Save Repository. To read a set of node settings from a node repository file, right-click and choose Open Repository. To back out all changes made on the Node Properties

page, reverting to the values deduced from the XML document, right-click and choose Clear.

Mapping Page

This page lets you specify the mapping between fields in the data packet and nodes in the XML document, create a transformation file, and save the transformation file.

The top of the page displays a two-column table that lists the nodes from the XML document and the corresponding fields in the data packet. When you first display the mapping page, this table is empty. In order to define a mapping, you must fill this table.

Note: You can only add nodes with values (text and attribute nodes), or nodes that have been marked as user-defined using the Node Properties

page.

Field View Page

This page displays the field attributes for all the fields in the data packet. Each node in the hierarchy represents a dataset, field, or field attribute:

Item	Description
▤	Represents the entire data packet or a dataset field. The children of a dataset node represent the fields in that dataset.
▤	Represents a field that is not a dataset field. The children of a field node represent the attributes of the field.
◆	Represents a field attribute, such as the data type, maximum length, and so on. The node is labeled with a string of the form AttributeName = Value, where AttributeName is the name of the field attribute and Value is its value.

Datapacket View Page

This page displays the structure of the data packet. The icons in this view are the same as those in the XML document pane, because data packets can be treated as special types of XML documents.

Item	Description
Ⓔ	Represents an element node. Element nodes in data packets represent datasets or dataset fields.
▤	Represents an attribute node. Attribute nodes in data packets represent fields (unless they are dataset fields).
▤	Represents a nested node. Nested nodes in data packets represent records.

Note: XML mapper can use a data packet in binary format (a .cds

file) as well as data packets that have been saved as XML. If you use a data packet in binary format, XML mapper converts it to XML format.

See Also

Using XML in Database Applications

3.2.16.12 Web App Debugger

Tools ▶Web App Debugger

Acts like a Web server on your development machine. If you build your Web server application as a Web App Debugger executable, deployment happens automatically during the build process. To debug your application, start it using **Run▶Run**. Next, select **Tools▶Web App Debugger**, click the default URL and select your application in the Web browser that appears. Your application will launch in the browser window, and you can use the IDE to set breakpoints and obtain debugging information.

See Also

Types of Web Server Applications

3.2.17 View

Topics

Name	Description
Add to Repository (↗ see page 1017)	Adds strings in the selected unit to the Translation Repository. This dialog is displayed when you right-click a node on the **Workspsace** tab of the Translation Manager and use the Add strings to repository comand. To add individual strings, rather than adding the strings for an entire unit, right-click the string in the Translation Manager and choose **Repository▶Add strings to repository**. The following options determine the criteria used for adding the strings.
Debug Windows (↗ see page 1018)	
Code Explorer (↗ see page 1033)	**View ▶Code Explorer** Navigates through the unit files. The Code Explorer contains a tree diagram that shows all of the types, classes, properties, methods, global variables, and global routines defined in your unit. It also shows the other units listed in the `uses` clause. Right-click an item in the Code Explorer to display its context menu. When you select an item in the Code Explorer, the cursor moves to that item's implementation in the Code Editor. When you move the cursor in the Code Editor, the highlight moves to the appropriate item in the Code Explorer. The Code Explorer uses... more (↗ see page 1033)
Customize Toolbars (↗ see page 1033)	**View▶Toolbars▶Customize** Changes the toolbar configuration. Using this dialog box, you can add, remove, and rearrange buttons on toolbars.
Data Explorer (↗ see page 1034)	**View▶Data Explorer** Adds new connections, modifies; deletes, or renames your connections. You can browse database server-specific schema objects including tables, fields, stored procedure definitions, triggers, and indexes. Additionally, you can drag and drop data from a data source to a project to build your database application quickly. The **Data Explorer** commands available depend upon the object selected in the tree view. Commands are available for the following nodes: • Provider types • Provider connections • Tables node • Individual tables • Individual views • Individual stored procedures
Delete Saved Desktop (↗ see page 1036)	**View ▶Desktops▶Delete Desktop** Delete a saved desktop by selecting it from the list and clicking **Delete**.
Desktop Toolbar (↗ see page 1036)	Selects an existing desktop layout or saves the current settings as a desktop layout. The **Desktop** toolbar is located at the far right end of the upper toolbar in the IDE.

File Browser (see page 1036)	**View ▶ File Browser** Dockable Windows-style File Browser views files and directories and performs simple operations on files while you are running the IDE. The File Browser supports standard Windows context menu options, as well as the following commands that are specific to RAD Studio:
Add to Repository (see page 1037)	Use this dialog box to add strings in the selected unit to the Translation Repository. This dialog is displayed when you right-click a node on the **Workspsace** tab of the Translation Manager and use the Add strings to repository comand. To add individual strings, rather than adding the strings for an entire unit, right-click the string in the Translation Manager and choose **Repository ▶ Add strings to repository**. The following options determine the criteria used for adding the strings.
Message View (see page 1038)	Displays messages such as compiler errors and warnings. You can copy one or more lines from the **Message** view to the clipboard. The **Build** tab displays the build command. The **Output** tab displays build output messages. To select the verbosity level for build output, use the **Tools ▶ Options ▶ Environment Options** page.
Object Inspector (see page 1038)	**View ▶ Object Inspector** Sets the properties and events for the currently selected object.
Project Manager (see page 1038)	**View ▶ Project Manager** Displays and organizes the contents of your current project group and any project it contains. You can perform several project management tasks, such as adding, removing, and compiling files. **Note:** Some features described here are available only in specific editions of the product. For example, some functionality in the Project Manager is available only for the C++ personality.
Save Desktop (see page 1047)	**View ▶ Desktops ▶ Save Desktop** Saves your current IDE desktop arrangement as a desktop layout.
Select Debug Desktop (see page 1047)	**View ▶ Desktops ▶ Set Debug Desktop** Determines which saved desktop layout is used when you are debugging.
Structure View (see page 1047)	**View ▶ Structure** Shows the hierarchy of source code or HTML displayed in the **Code Editor**, or components displayed on the **Designer**. When displaying the structure of source code or HTML, you can double-click an item to jump to its declaration or location in the **Code Editor**. When displaying components, you can double-click a component to select it on the form. If your code contains syntax errors, they are displayed in the **Errors** node in the **Structure View**. You can double-click an error to locate the corresponding source in the **Code Editor**. ((Not applicable for... more (see page 1047)
Templates Window (see page 1048)	**View ▶ Templates** Creates, edits, or deletes live code templates.
To-Do List (see page 1049)	**View ▶ To-Do List** Creates and manages a to-do list.
Add or Edit To-Do Item (see page 1049)	Adds items to a to-do list or to change an item.
Filter To-Do List (see page 1050)	Controls which items are displayed in a to-do list.
Table Properties (see page 1050)	Controls the appearance of the resulting to-do list when using the **Copy as ▶ HTML table** command from the right-click menu of the **To-do List** dialog.
Tool Palette (see page 1051)	**View ▶ Tool Palette** Assists with a new project, adds components to a form, or adds code snippets to the **Code Editor**.
Translation Manager (see page 1052)	**View ▶ Translation Manager** Views and edits language resource files.
Multi-line Editor (see page 1054)	Edits translations that are lengthy or contain multiple lines of text separated by hard returns. The editor displays the source and target languages in separate panes. Only the target language is editable.
Type Library Editor (see page 1054)	**View ▶ Type Library** Makes changes to your type library. The Type Library editor generates the required IDL syntax automatically. Any changes you make in the editor are reflected in the corresponding implementation class (if it was created using a wizard). The **View ▶ Type Library** command is available only for projects that contain a type library. The wizards on the ActiveX page automatically add a type library to the project when they create a COM object.

View Form (see page 1057)	**View ▶ Forms** Views any form in the current project. When you select a form, it becomes the active form, and its associated unit becomes the active module in the **Code Editor**.
View Units (see page 1057)	**View ▶ Units** Views the project file or any unit in the current project. When you open a unit, it becomes the active page in the **Code Editor**.
Window List (see page 1057)	**View ▶ Window List** Displays a list of open windows.
New Edit Window (see page 1057)	**View ▶ New Edit Window** Brings up a new **Code Editor** window as a separate window. The previous **Code Editor** window remains open.
Toggle Form/Unit (see page 1058)	**View ▶ Toggle Form/Unit** Toggles the view between Form and Unit.
Model View Window (see page 1058)	**View ▶ Model View** Shows the logical structure and containment hierarchy of your project. Note the ECO framework is available only for C# and Delphi for .NET, and in the Architect SKU and higher. ECO-related icons and topic links are unavailable in other product SKUs.
CodeGuard Log (see page 1059)	**View ▶ Debug Windows ▶ CodeGuard Log** Provides runtime debugging for C++ applications being developed. CodeGuard reports errors that are not caught by the compiler because they do not violate the syntax rules. CodeGuard tracks runtime libraries with full support for multithreaded applications.
Desktops (see page 1059)	**View ▶ Desktops** Allows you to choose between preset desktop layouts. Desktop layouts can be used to create and manage windows.
Dock Edit Window (see page 1060)	**View ▶ Dock Edit Window** Sizes new **Code Editor** windows to fit appropriately inside the IDE. You can reselect Dock Edit Window to toggle between the new **Code Editor** window and the original **Code Editor** window.
Find Reference Results (see page 1060)	**View ▶ Find Reference Results** Brings up the **Find References** pane. This pane is dockable and is used in conjunction with the **Search ▶ Find** function.
Help Insight (see page 1060)	**View ▶ Help Insight** Displays a hint containing information about the symbol such as type, file, location of declaration, and any XML documentation associated with the symbol (if available). Alternative ways to invoke Help Insight is to hover the mouse over an identifier in your code while working in the **Code Editor**, or by pressing `CTRL+SHIFT+H`.
Show Borders (see page 1060)	**View ▶ Show Borders** Displays gray borders that represent page margins in the Diagram View and Overview. Diagrams exist within the context of a namespace (or a package). This feature is only applies to ASP .NET applications.
Show Grid (see page 1061)	**View ▶ Show Grid** Shows the design grid in the background behind diagrams. Diagrams exist within the context of a namespace (or a package). This feature is only applies to ASP .NET applications. To turn on the grid, open **Tools ▶ Options ▶ HTML/ASP.NET**. In the **Designer Options** select Grid Layout from the pull-down menu.
Show Tag Glyphs (see page 1061)	**View ▶ Show Tag Glyphs** Displays tags in an ASP form. This feature only applies to ASP .NET applications.
Snap To Grid (see page 1061)	**View ▶ Snap To Grid** Allows diagram elements to "snap" to the border of a control to the nearest coordinate of the diagram background design grid. The snap function works whether the grid is visible or not. Diagrams exist within the context of a namespace (or a package). This feature is only applies to ASP .NET applications.
Toolbars (see page 1061)	**View ▶ Toolbars** Allows you to choose the toolbars that are displayed in the IDE.
Translation Editor (see page 1062)	**View ▶ Translation Manager ▶ Translation Editor** Edits resource strings directly, adds translated strings to the Translation Repository, or gets strings from the Translation Repository.

Welcome Page (see page 1062)	**View ▶Welcome Page** Opens the product's Welcome Page, which displays lists of your recent projects and favorites. The Welcome Page also contains links to developer resources, such as product-related online help. As you develop projects, you can quickly access them from the list of recent projects at the top of the Welcome Page.

3.2.17.1 Add to Repository

Adds strings in the selected unit to the Translation Repository. This dialog is displayed when you right-click a node on the **Workspsace** tab of the Translation Manager and use the Add strings to repository comand.

To add individual strings, rather than adding the strings for an entire unit, right-click the string in the Translation Manager and choose **Repository▶Add strings to repository**.

The following options determine the criteria used for adding the strings.

Item	Description
Status	Adds strings based on the status displayed in the **Status** column on the **Workspace** tab. Check the statuses that you want to add.
Duplicate action	Determines how the repository responds when it finds a duplicate translation string for the same source string. **Skip** does not add the string. **Add** adds the string to the repository if no translated string exists for the original string. **Force Add** always adds the string to the repository, regardless of whether it exists in the repository. **Replace** overwrites the existing string with new string. **Display selection** offers the user a choice.
Include context information	Adds the unit path, and the value displayed in the **Id** column of the **Workspace** tab, to the Translation Repository. This context information is displayed in the status bar when you select a string in the Translation Repository.
Value	Indicates whether a string is added based on changes to its original value. **Changed** adds the string only if the original and translated values are different. **Unchanged** adds the string even if the original and translated values are the same. **Don't care** adds the string, whether it has changed or not, provided the string meets the other criteria set in this dialog box.
Comment	Adds or excludes strings based on the text in the **Comment** column on the **Workspace** tab. Type the comment text in the edit box and check **Include** to add strings with a matching comment, or check **Not include** to exclude strings with a matching comment.

Tip: To set general options for the Translation Repository, choose Tools->Translation Tools Options

and select **Repository**.

See Also

Localizing Applications (see page 18)

3.2.17.2 Debug Windows

Topics

Name	Description
Add/Edit Module Load Breakpoint (see page 1019)	**Run ▶ Add Breakpoint ▶ Module Load** Adds a module load breakpoint that will halt program execution when the module is loaded. You also use this dialog when you click Edit Type the module name (usually a DLL or package) into the edit box, or click the **Browse** button to navigate to the module. This dialog box also adds the module to the **Modules** window. Modules are automatically added to the **Modules** window when they are loaded into memory, but if you want to halt execution for debugging when the module first loads into memory, you must add it to the modules... more (see page 1019)
Add Watch Group (see page 1019)	Enters a name for a new watch group, or selects a name from the list of previously entered watch group names. The watch group will be added as a tab in the **Watch List**.
Breakpoint List Window (see page 1019)	**View ▶ Debug Windows ▶ Breakpoints** Displays, enables, or disables breakpoints currently set in the loaded project. Also changes the condition, passes count, or groups associated with a breakpoint. If no project is loaded, it shows all breakpoints set in the active **Code Editor** or in the **CPU** window. **Tip:** Several items on the Breakpoint List context menu are also available on the **Breakpoint List** toolbar.
Call Stack Window (see page 1021)	**View ▶ Debug Windows ▶ Call Stack** Displays the function calls that brought you to your current program location and the arguments passed to each function call. The **Call Stack** window lists the last function called, followed by each previously called function. The first function called is at the bottom of the list. If debug information is available for a function, it is followed by the arguments that were passed when the call was made. Double-clicking an item in the **Call Stack** window displays both the source for the frame and the locals for the frame. To toggle a breakpoint on a... more (see page 1021)
CPU Window (see page 1022)	**View ▶ Debug Windows ▶ CPU Windows** Displays the assembly language code for the program you are debugging. This window opens automatically when program execution stops at a location for which source code is unavailable. The **CPU** window is divided into the following panes:
Enter New Value (see page 1025)	Modifies the value located at the current cursor position in the **CPU** or **FPU** views. This dialog is displayed when you right-click and choose Change from the **Dump**, **Stack**, or **Register** pane of the **CPU** window or the **Register** pane of the **FPU** window. Enter a value for the currently selected item. Precede hexadecimal values with $. From the **Dump** and **Stack** panes of the **CPU** window, you can enter more than one value separated by a space. You must enter a value that corresponds to the current display type set using Display As. From the **Register** pane... more (see page 1025)
Enter Search Bytes (see page 1025)	Searches forward in the **Disassembly** pane of the **CPU** window for an expression or byte list.
Event Log Window (see page 1025)	**View ▶ Debug Windows ▶ Event Log** Shows messages for breakpoints, process control, threads, modules, and output that occur during a debug session. You can copy text in the **Event Log** by using `Ctrl-C` or the Edit->Copy command. Right-click the **Event Log** window to display the following commands. **Note:** To copy Event Log messages to the clipboard, use **Edit ▶ Copy** or `Ctrl-C`.
Add Comment to Event Log (see page 1026)	Adds a comment to the end of the **Event Log** window.
FPU (see page 1026)	**View ▶ Debug Windows ▶ FPU** Displays the contents of the Floating-point Unit and SSE registers in the CPU.
Local Variables Window (see page 1028)	**View ▶ Debug Windows ▶ Local Variables** Shows the current function's local variables while in debug mode. To view local variables from a non-current stack frame, select a frame from the drop-down list. Right-click the **Local Variables** window to display the following commands.

Modules Window (↗ see page 1028)	**View ▶ Debug Windows ▶ Modules** Shows processes under control of the debugger and the modules currently loaded by each process. This window is divided into the following areas.
Source File Not Found (↗ see page 1030)	Locates a file that the debugger can not find.
Threads (↗ see page 1030)	**View ▶ Debug Windows ▶ Threads** Shows the status of all processes and threads that are executing in each application being debugged.
Watch List Window (↗ see page 1031)	**View ▶ Debug Windows ▶ Watch List** Displays the current value of the watch expression based on the scope of the execution point. The **Watch List** window is a multi-tabbed view with each tab representing a distinct watch group. Only the watch group on the active tab is evaluated while debugging. **Tip:** To enable or disable a watch expression quickly, use the check box ☑ next to the watch.
Disassembly (↗ see page 1032)	**View ▶ Debug Windows ▶ CPU Windows ▶ Disassembly** Displays the address, the hexadecimal representation of the machine code instructions (opcodes), and the assembly instructions for each line of source code. The address is the offset into the disassembled method.
Memory (↗ see page 1032)	**View ▶ Debug Windows ▶ CPU Windows ▶ Memory** Displays the raw values contained in addressable areas of your program. Displayed only for unmanaged code. The pane displays the memory addresses, the current values in memory, and an ASCII representation of the values in memory. There are four different views in order to view four distinct areas in memory at the same time. You can have more than one view and then use **Search ▶ Goto Address** in the separate views to look at different places in memory at the same time.
Registers (↗ see page 1032)	**View ▶ Debug Windows ▶ CPU Windows ▶ Registers** Displays the contents of the CPU registers of the 80386 and greater processors.
Stack (↗ see page 1032)	**View ▶ Debug Windows ▶ CPU Windows ▶ Stack** Displays the raw values contained in the program stack. Displayed only for unmanaged code.

3.2.17.2.1 Add/Edit Module Load Breakpoint

Run ▶ Add Breakpoint ▶ Module Load

Adds a module load breakpoint that will halt program execution when the module is loaded. You also use this dialog when you click Edit Type the module name (usually a DLL or package) into the edit box, or click the **Browse** button to navigate to the module.

This dialog box also adds the module to the **Modules** window. Modules are automatically added to the **Modules** window when they are loaded into memory, but if you want to halt execution for debugging when the module first loads into memory, you must add it to the modules window.

See Also

Modules Window (↗ see page 1028)

3.2.17.2.2 Add Watch Group

Enters a name for a new watch group, or selects a name from the list of previously entered watch group names.

The watch group will be added as a tab in the **Watch List**.

3.2.17.2.3 Breakpoint List Window

View ▶ Debug Windows ▶ Breakpoints

Displays, enables, or disables breakpoints currently set in the loaded project. Also changes the condition, passes count, or groups associated with a breakpoint. If no project is loaded, it shows all breakpoints set in the active **Code Editor** or in the **CPU** window.

Tip: Several items on the Breakpoint List

context menu are also available on the **Breakpoint List** toolbar.

Column	Description
☑	Indicates whether the breakpoint is enabled or disabled. Check the box to enable the breakpoint. Uncheck it to disable the breakpoint.
Filename/Address	The source file for the source breakpoint or the address for the address breakpoint.
Line/Length	The code line number for the breakpoint or the length (the number of bytes to watch) for the data breakpoint.
Condition	The conditional expression that is evaluated each time the breakpoint is encountered. Click a condition value to edit it.
Action	The action associated with breakpoints.
Pass Count	The current pass and the total number of passes specified for the breakpoint. Click a pass count value to edit it.
Group	The group name with which the breakpoint is associated. Click a group value to edit it.

The following icons are used to represent breakpoints in the **Breakpoint List** window.

Icon	Description
●	The breakpoint is valid and enabled.
○	The breakpoint is valid and disabled.
⊗	The breakpoint is set at an invalid location, such as a comment, a blank line, or invalid declaration.

Context Menu if No Breakpoint is Selected

Right-click the **Breakpoint List** window (not on an actual breakpoint) to display the following commands:

Item	Description
Add	Opens dialog boxes where you can create new breakpoints.
Delete All	Removes all breakpoints. This command is not reversible.
Disable All	Disables all enabled breakpoints. When you disable a breakpoint, its settings remain defined, but the breakpoint does not cause your program to stop. Disabling is useful when you temporarily do not need a breakpoint but want to preserve its settings.
Enable All	Enables all disabled breakpoints.
Disable Group	Disables the breakpoint group that you select.
Enable Group	Enables the breakpoint group you select.
Dockable	Toggles whether the **Breakpoint List** window is dockable.
Stay On Top	Keeps the window visible when out of focus.

Context Menu if a Breakpoint is Selected

Right-click on a breakpoint to display the following commands:

Item	Description
Enabled	Toggles between enabling and disabling a breakpoint.
Delete	Removes a breakpoint.
View Source	For source breakpoints, locates a breakpoint in your source code. For address breakpoints, displays the location in the **CPU** window.
Edit Source	For source breakpoints, locates a breakpoint in your source code and activates the **Code Editor**. For address breakpoints, displays the location in the **CPU** window.
Properties	Displays the **Breakpoint Properties** dialog box, where you can modify breakpoints.
Breakpoints	Displays a menu of breakpoint commands.
Stay On Top	Keeps the window visible when out of focus.
Dockable	Enables drag-and-dock for the **Breakpoint List** window.

3.2.17.2.4 Call Stack Window

View ▶ Debug Windows ▶ Call Stack

Displays the function calls that brought you to your current program location and the arguments passed to each function call. The **Call Stack** window lists the last function called, followed by each previously called function. The first function called is at the bottom of the list. If debug information is available for a function, it is followed by the arguments that were passed when the call was made.

Double-clicking an item in the **Call Stack** window displays both the source for the frame and the locals for the frame.

To toggle a breakpoint on a particular frame, either click the breakpoint icon in the far left column, or right-click the frame and click the Toggle Breakpoint command on the context menu. The icon in the far left column of the Call Stack window indicates the following:

- A blue arrow (➡) indicates the top stack frame.
- A red checkmark (✓) indicates there is an enabled breakpoint set on a frame.
- A grey checkmark (✓) indicates there is a disabled breakpoint set on a frame.
- A blue circle (●) indicates that the frame has debug information (symbols are available).
- A grey circle (●) indicates that the frame has no debug information (no symbols are available).

Item	Description
View Source	Scrolls the Code editor to the location of the function call that is selected in the **Call Stack** window, but does not give the Code editor focus.
Edit Source	Scrolls the **Code Editor** window to the location of the function call that is selected in the **Call Stack** window, and sets the focus to the **Code Editor**.
View Locals	Displays in the **Local Variables** window any local variables associated with function call currently selected in the **Call Stack** window.
Toggle Breakpoint	Sets a breakpoint that is disabled, or disables a breakpoint that is set. Breakpoint icons are displayed in colors that indicate the symbols status of the frame. See the description of icon colors preceding this list.
Show Fully Qualified Names	Displays full paths of file names.
Stay On Top	Keeps the **Call Stack** window visible when out of focus.

Dockable	Enables drag-and-dock for **Call Stack** window.

See Also

Compiling (⧉ see page 2)

Setting and Modifying Source Breakpoints (⧉ see page 118)

3.2.17.2.5 CPU Window

View ▸ Debug Windows ▸ CPU Windows

Displays the assembly language code for the program you are debugging. This window opens automatically when program execution stops at a location for which source code is unavailable.

The **CPU** window is divided into the following panes:

Area	**Description**
Address Status (at the top of the window)	Displays the effective address (when available) and the value stored at that address. For example, if you select an address containing an expression in brackets such as [eax+edi*4-0x0F], the location in memory being referenced and its current value is displayed. The current thread ID is also displayed.
Disassembly pane (upper left side)	Displays the address, the hexadecimal representation of the machine code instructions (opcodes), and the assembly instructions for each line of source code. The address is the offset into the disassembled method. If you are debugging managed code, the assembly instructions correspond to the native code created by the JIT compiler. The Microsoft Intermediate Language (MSIL) created by the compiler is also displayed. Note that there is not a one-to-one relationship between the native code instructions and the MSIL instructions. You can not step into or set breakpoints on the MSIL instructions. If debug information is available, the debugger displays the source code that corresponds to the assembly instructions. A right arrow (➔) to the left of an address indicates the current execution point. When the current instruction is a transfer instruction (for example, `call` or `jmp`), either an up or down arrow after the instruction indicates the target direction for the transfer instruction. For example, if the target is located before the current instruction, an up arrow is displayed. If the target is after the current instruction, a down arrow is displayed. For conditional transfer instructions (for example, `jz` or `jle`), an arrow is displayed only if the condition is true. For conditional set instructions (for example, `seta` or `setz`), a left arrow is displayed if the condition is true.
Register pane (upper middle pane)	Displays the contents of the CPU registers of the 80386 and greater processors. These registers consist of eight 32-bit general purpose registers and the 32-bit program counter (EIP). When debugging Win32 code, the flags (EFL) register and the six segment registers are also displayed. After you execute an instruction, any registers that have changed value since the program was last paused are highlighted in red.
Memory Dump pane (lower left side)	Displayed only for unmanaged code. Displays the raw values contained in addressable areas of your program. The pane displays the memory addresses, the current values in memory, and an ASCII representation of the values in memory. The leftmost part of each line shows the starting address of the line. Following the address listing is an 8-byte hexadecimal listing of the values contained at that location in memory. Each byte in memory is represented by two hexadecimal digits. Following the hexadecimal display is an ASCII display of the memory. Non-printable values are represented with a period. Press `CTRL+LEFT ARROW` or `CTRL+RIGHT ARROW` to shift the starting point of the display up or down one byte.

Flags pane (upper right side)	Displayed only for unmanaged code. Displays the current state of the flags and information bits contained in the 32-bit register EFL. After you execute an instruction, the **Flags** pane highlights in red any flags that have changed value since the program was last paused. The processor uses the bits in this register to control certain operations and indicate the state of the processor after it executes certain instructions. Pass the mouse over a flag to display the flag name.
Machine Stack pane (lower right side)	Displayed only for unmanaged code. Displays the raw values contained in the your program stack. The pane has three sections: the memory addresses, the current values on the stack, and an ASCII representation of the stack values. A green arrow indicates the value at the top of the call stack.

Single Panes of the CPU Window Are Dockable

You can now open a single pane of the **CPU** window (such as the **Disassembly**, **Registers**, or **Stack** views), from the **View ▶ Debug Windows** submenu. A single pane becomes a dockable view that you can move around inside the IDE.

CPU Window Automatically Closes

If **Automatically close files implicitly opened while debugging** is checked on the **Tools ▶ Options ▶ Debugger Options** window, the **CPU** window automatically closes when you end your debugging session. However, if the **CPU** window is the top window, it does not close.

Scrolling the Disassembly Pane

Use any of the following methods to scroll the **Disassembly** pane:

- Press CTRL+LEFT ARROW and CTRL+RIGHT ARROW to shift the starting point of the display up or down one byte. Changing the starting point of the display changes where the debugger begins disassembling the machine code.
- Click above or below the vertical scrollbar to scroll up or down a screen. (Due to the high volume of information available for display in the **Disassembly** pane, dragging the scrollbar is disabled.)
- Use the Goto Address, Goto Current EIP, Follow, and Previous context menu commands, as described in the following section.

Context Menu

The following table lists alphabetically the commands for the panes in the **CPU** window. Right-click **CPU** window to display the following context menu commands.

Item	Description
Breakpoint Properties	Displays the **Address Breakpoint Properties** dialog box.
Caller	Positions the **Disassembly** pane to the instruction past the one that called the current interrupt or subroutine. If the current interrupt routine has pushed data items onto the stack, the debugger might not be able to determine where the routine was called from. **Caller** works best when you turn on **Stack frames** option under **Code Generation** (on the **Project ▶ Options ▶ Compiler** page).
Change	Lets you modify the bytes located at the current cursor location and prompts you for an item of the current display type.
Change register	Displays the **Change Register** dialog box where you enter a new value for the register. You can make full use of the expression evaluator to enter new values. Be sure to precede hexadecimal values with $.
Change thread	Displays the **Select a Thread** dialog box, where you can select the thread you want to debug from the threads listed. When you choose a new thread from the **Flags** pane, all panes in the **CPU** window reflect the state of the CPU for that thread.
Copy	Copies all selected instructions to the clipboard. From the disassembly pane, you can select a single instruction or you can use the SHIFT key to select multiple instructions. In all other panes, you can only select a single item to copy.

Decrement register	Subtracts 1 from the value in the currently highlighted register. This option lets you test “off-by-one” bugs by making small adjustments to the register values.
Display as	Formats the data listed in the **Machine Stack** pane of the **CPU** window. Choose from the following formats: **Data type** displays format. **Bytes** displays data in hexadecimal bytes. **Words** displays data in 2-byte hexadecimal numbers. **DWords** displays data in 4-byte hexadecimal numbers. **Singles** displays data in 4-byte floating-point numbers using scientific notation.
Enabled	Available only when right-clicking a breakpoint. Toggles the breakpoint between enabled and disabled.
Follow	Positions the pane at the destination address of the currently instruction highlighted.
Goto Address	Displays the **Enter Address to Position** dialog box where you can enter a symbol or, for managed code, an address in just in time (JIT) compiler format.
Goto Current EIP	Positions the **CPU** window to the location of the current program counter (the location indicated by the EIP register). This location indicates the next instruction to be executed by your program.
Increment register	Adds 1 to the value in the currently highlighted register. This option lets you test "off-by-one” bugs by making small adjustments to the register values.
Mixed IL Code	When debugging managed code, toggles the display to include MSIL instructions.
Mixed Source	Toggles the display between assembly instructions only and assembly instructions and their corresponding source code (if debug information is available).
New EIP	Changes the location of the instruction pointer (the value of EIP register) to the line currently highlighted in the **Disassembly** pane. Use this command when you want to skip certain machine instructions. When you resume program execution, execution starts at this address. This command is not the same as stepping through instructions; the debugger does not execute any instructions that you might skip. Use this command with extreme care; it is easy to place your system in an unstable state when you skip over program instructions.
Next	Finds the next occurrence of the item you last searched for in the **Memory Dump** pane.
Previous	Restores the **CPU** window to the display it had before the last Follow command.
Run to Current	Runs your program at full speed to the instruction that you have selected in the **CPU** window. After your program is paused, you can use this command to resume debugging at a specific program instruction.
Search	Displays the **Enter Search Bytes** dialog box where you can search forward in the **CPU** window for an expression or byte list (click **Help** on the **Enter Search Bytes** dialog box for details).
Show addresses	Includes instruction addresses.
Show opcodes	Includes instruction opcodes. Choices are **Auto**, **Always**, and **Never**. **Auto** is the default value and causes opcodes to be shown whenever the window is wide enough to contain the opcode column.
Toggle Breakpoint	Set or removes a breakpoint at the currently selected address.
Toggle flag	The flag and information bits in the **Flags** pane can each hold a binary value of 0 or 1. This command toggles the selected flag or bit between these two binary values.
Top of stack	Positions the **Machine Stack** pane at the address of the stack pointer (the address held in the ESP register).
View FPU	Available only when debugging Win32 code. Displays the **FPU** view, which displays the floating-point registers, MMX registers, and SSE registers.

View Source	Activates the **Code Editor** and positions the insertion point at the source code line that most closely corresponds to the disassembled instruction selected in the **CPU** window. If there is no corresponding source code, this command has no effect.
Zero register	Sets the value of the currently highlighted register to 0.

3.2.17.2.6 Enter New Value

Modifies the value located at the current cursor position in the **CPU** or **FPU** views.

This dialog is displayed when you right-click and choose Change from the **Dump**, **Stack**, or **Register** pane of the **CPU** window or the **Register** pane of the **FPU** window. Enter a value for the currently selected item. Precede hexadecimal values with $.

From the **Dump** and **Stack** panes of the **CPU** window, you can enter more than one value separated by a space. You must enter a value that corresponds to the current display type set using Display As.

From the **Register** pane of the **FPU** view, specify a single 32-bit hexadecimal value (use of decimal numbers is allowed but is not typical).

3.2.17.2.7 Enter Search Bytes

Searches forward in the **Disassembly** pane of the **CPU** window for an expression or byte list.

Item	**Description**

Edit Box	Enter a byte list for two or more values located in a specific order. Precede hexadecimal values with `0x`. For example, if you enter `0x5D 0xC3`, the debugger goes to the following location:
	`00000001 5D`
	`00000002 C3`
	Alternatively, you can use a dollar sign (`$`) instead of `0x`.
	To search for DWords, reverse the order of the bytes. For example, if you enter `0X1234,` the debugger positions the pane to memory location `34 12`.

3.2.17.2.8 Event Log Window

View ▶ Debug Windows ▶ Event Log

Shows messages for breakpoints, process control, threads, modules, and output that occur during a debug session.

You can copy text in the **Event Log** by using `Ctrl-C` or the Edit->Copy command.

Right-click the **Event Log** window to display the following commands.

Note: To copy Event Log

messages to the clipboard, use **Edit ▶ Copy** or `Ctrl-C`.

Item	**Description**
Clear Events	Removes all messages from the **Event Log** window.
Save Events to File	Displays the **Save Event Log to File** dialog box, allowing you to save the messages in the **Event Log** window to a text file.
Add Comment	Displays the **Add Comment to Event Log** dialog, allowing you to add a comment to the end of the event log.

Properties	Displays the **Debugger Event Log Properties** page, allowing you control the content and appearance of the **Event Log** window. You can also display this page by choosing **Tools ▶ Options ▶ Debugger Options ▶ Event Log**.
Stay On Top	Keeps the window visible when out of focus.
Dockable	Enables drag-and-dock for the **Event Log** window.

3.2.17.2.9 Add Comment to Event Log

Adds a comment to the end of the **Event Log** window.

Item	Description
Comment	Enter the comment you want to appear in the **Event Log** window.

3.2.17.2.10 FPU

View ▶ Debug Windows ▶ FPU

Displays the contents of the Floating-point Unit and SSE registers in the CPU.

Item	Description
Instruction Pointer (IPTR)	Displays the Instruction Pointer (IPTR) address, opcode, operand (OPTR) address of the last floating-point instruction executed.
FPU Registers pane	Displays the floating-point register stack (ST0 through ST7) in ascending order. After the list, the control word, status word, and tag word are shown. The information displayed for each of the eight registers is shown as follows: Register name, register status, and register value. The register status can be one of the following values: **Empty** Indicates that the register contains invalid data. When a register is empty, no value is displayed for that register, because the data in the register is presumed to be invalid. **Valid** Indicates that the register contains nonzero valid data. **Spec.** (Special) Indicates that the register contains valid data, but the valid data represents a special condition, either NAN (not a number), infinity, or a denormalized value. The status of each register is determined by examining the tag word and the eleventh through thirteenth bits of the status word (top of stack indicator). When a register's status is not Empty, the value of the register in long double (extended) format is displayed immediately following the status. The registers can be displayed in different formats (other than long doubles). The control, status, and tag words are displayed in hexadecimal format only. For these three words, any values that were altered by the last run operation are displayed in red.

Control Flags pane	Lists the control flags encoded in the control word. Any flags that were altered by the last run operation are displayed in red. The control flags and their bit number in the control word are as follows: **IM** Invalid Operation Exception, 0 **DM** Denormalized Operation Exception Mask, 1 **ZM** Zero Divide Exception Mask, 2 **OM** Overflow Exception Mask, 3 **UM** Underflow Exception Mask, 4 **PM** Precision Exception Mask, 5 **PC** Precision Control, 8, 9 **RC** Rounding Control, 10, 11 **IC** Infinity Control (Obsolete), 12 Select any of the flags and right-click to change the flag’s value. For single-bit flags, it changes the value from 0 to 1 or from 1 to 0. For multi-bit flags, it cycles through all possible values.
Status Flags pane	Lists the status flags encoded in the status word. Any flags that were altered by the last run operation are displayed in red. The flags and their bit number in the control word are as follows: **IE** Invalid Operation Exception, 0 **DE** Denormalized Operation Exception, 1 **ZE** Zero Divide Exception, 2 **OE** Overflow Exception, 3 **UE** Underflow Exception, 4 **PE** Precision Exception, 5 **SF** Stack Fault, 6 **ES** Error Summary Status, 7 **C0** Condition Code 0 (CF), 8 **C1** Condition Code 1, 9 **C2** Condition Code 2 (PF), 10 **ST** Top of Stack, 11-13 **C3** Condition Code 3 (ZF), 14 **BF** FPU Busy, 15 Select any of the flags and right-click to change the flag’s value. For single-bit flags, it changes the value from 0 to 1 or from 1 to 0. For multi-bit flags, it cycles through all possible values.
SSE pane	Displays the Streaming SIMD Extensions (SSE) registers. Right-click the SSE pane and choose Display As to change the display format of the register content.

Context Menu

Right-click **FPU** window to display the following context menu commands.

Item	**Description**
Zero	Sets the selected register's value to 0. When used on one of the seven FPU registers, this command also sets that register's tag bits in the tag word to 01 indicating that the register holds a zero value.
Empty	Sets the selected register's tag bits in the tag word to 11 indicating that the register is empty. This command is grayed out if the selected register is the CTRL word, STAT word, or TAG word.

Change	Displays the **Change** dialog, where you can enter a new value for the selected register. When used on one of the seven FPU registers, this command also sets that register's tag bits in the tag word to 00 indicating that the register holds a valid value. The value you enter in the **Change** dialog should match the format currently specified by the **Display As** menu item. For example, if the currently displayed format is **Extended**, you should enter an Extended value in the **Change** dialog.
Display As	Determines how the values in registers are displayed. The items on the submenu change depending on Show menu selection. For FPU registers, the possible display types are Words and Extendeds (long doubles). For MMX registers, the possible display types are Bytes, Words, DWords (double words), and QWords (quad words). For SSE registers, the possible values are Bytes, Words, DWords (double words), QWords (quad words), DQWords (double quad words), Singles, and Doubles.
Radix	Available only when MMX registers are shown. Determines how the values in the MMX register are displayed. The possible values are Binary, Unsigned Decimal, Hexadecimal, and Signed Decimal.
Show	Toggles the **FPU Registers** pane between between FPU and MMX registers: **Floating Point Registers** displays the 10-byte FPU registers ST(0) through ST(7). The registers can be viewed as either Extended (long double) values or as 5 DWord values. **MMX Registers** displays the 8-byte MMX registers MM0 through MM7. The registers can be viewed as 8 Byte values, 4 Word values, 2 DWord values, or 1 QWord value. These values can be shown in either binary, decimal, or hexadecimal format (see **Radix**). MMX registers can only be shown on a computer that is MMX enabled.
Toggle Flag	In the **Status Flags** and **Control Flags** panes, changes the value of the selected flag. For single-bit flags, changes the value from 0 to 1 or from 1 to 0. For multi-bit flags, cycles through all possible values.
Stay on Top	Keeps the **FPU** view on top of other windows.

3.2.17.2.11 Local Variables Window

View ▶Debug Windows▶Local Variables

Shows the current function’s local variables while in debug mode. To view local variables from a non-current stack frame, select a frame from the drop-down list.

Right-click the **Local Variables** window to display the following commands.

Item	Description
Inspect	Displays information about the currently selected variable in the **Inspector** window.
Stay On Top	Keeps the **Local Variables** window visible, even when it does not have focus.
Dockable	Enable drag-and-dock for the **Local Variables** window.

Tip: You can display this window by pressing CTRL+ALT+L

while any IDE window has focus, even if you are not in debug mode. However, the window will be empty unless the debugger is paused. Keep this window open during your debugging sessions to monitor how your program updates the values of variables as the program runs.

3.2.17.2.12 Modules Window

View▶Debug Windows▶Modules

Shows processes under control of the debugger and the modules currently loaded by each process. This window is divided into the following areas.

Area	Description
Modules pane (upper left side)	Displays the processes and modules sorted, by default, in the order in which they are loaded. Each process can have one or more modules which it loads. When a process terminates or a module is unloaded, it is removed from the list. ➡ indicates the current process. To sort the display, click a column heading.
Source pane (lower left side)	If debug information is available, displays the source files that were used to build the module currently selected in the **Modules** pane.
Scope Browser (right side, for managed code only)	Displays a hierarchical tree view of the namespaces, classes, and methods used in the application. 📁 represents a namespace. ● represents a class. ⇄ represents a method.
Entry Point pane (right side, for unmanaged code only)	Displays the name and addresses of the entry points for the module currently selected in the **Modules** pane. The entry point is only shown if the source for it can found. To sort the display, click a column heading. The runtime image base address is the memory offset, in hexadecimal, where the module actually loads, and is distinct from the preferred image base address you may have specified in the **Project** ▶ **Options** window.

Context Menus

Right-click the **Modules** pane to display the following commands for unmanaged code.

Item	Description
Break On Load	Toggles a breakpoint to halt the execution of the application when it loads the selected module into memory. This setting is used only by the Borland Win32 Debugger. You can also click on the module icon to toggle a module load breakpoint.
Reload Symbol Table	Displays the **Reload Symbol Table** dialog box, allowing you to load the debug symbol table into the **Modules** window.
Add Module	Displays the **Add Module** dialog, allowing you to add a module to the list. Use this command to add a module load breakpoint on a module that is not currently loaded. This setting is used only by the CodeGear Win32 Debugger.

Right-click the **Source** pane to display the following command.

Item	Description
Edit Source	Activates the **Code Editor** and positions it to the selected module.

Right-click the **Scope Browser** to display the following commands (the **Scope Browser** is only displayed in managed debugging).

Item	Description
Browse Class	Displays the **CodeGear Reflection** tool, allowing you to inspect the currently selected class.
Edit Source	Available for methods only. Activates the **Code Editor** and positions it to the method. Selecting a method that has not been Just In Time compiled yet results in the message `No native code is available`.

Right-click the **Entry Point** pane to display the following command.

Item	Description
Go to Entry Point	Displays the selected entry point in the **CPU** window if there is no source for the entry point. If source can be found, it will be shown. Your program must be paused before you can jump to an entry point.

3.2.17.2.13 Source File Not Found

Locates a file that the debugger can not find.

Item	Description
Path to source file	Displays the name of the source file that the debugger can not find. Click the **Browse** button to navigate to the source file or type the full path name of the source file.
Add directory to Debug Source Path	Appends the file path to the end of the debug source path (in **Project** ▶ **Options** ▶ **Debugger**).

3.2.17.2.14 Threads

View ▶ **Debug Windows** ▶ **Threads**

Shows the status of all processes and threads that are executing in each application being debugged.

Item	Description
Thread ID	Displays the process name, the OS assigned thread ID, and, if the thread is named, its name.
State	Indicates the execution state of the thread as Runnable, Stopped, Blocked, or None; for processes, the state indicates how the process was created: Spawned, Attached, or Cross-process Attach.
Status	Indicates the thread status as one of the following: **Breakpoint** - The thread stopped due to a breakpoint. **Faulted** - The thread stopped due to a processor exception. **Unknown** - The thread is not the current thread so its status is unknown. **Stepped** - The last step command was successfully completed.
Location	Indicates the function name or address associated with the thread.

Tip: The current process is marked with a green arrow. Non-current processes are marked with a light blue arrow.

The current process and current thread become the context for the next user action, for example, run, pause, or reset.

Context Menu

Right-click the **Thread Status** window to display the following commands.

Item	Description
View Source	Displays the **Code Editor** at the corresponding source location of the selected thread ID, but does not make the **Code Editor** the active window.
Go to Source	Displays the **Code Editor** at the corresponding source location of the selected thread ID and makes the **Code Editor** the active window.
Make Current	Makes the selected thread the active thread if it is not so already. If the thread is not already part of the active process, its process also becomes the active process.
Terminate Process	Terminates the process, if a process is selected, or the process that the thread is part of, if a thread is selected.

Detach Process	Detaches the process, if a process is selected, or the process that the thread is part of, if a thread is selected..
Pause Process	Pauses the process, if a process is selected, or the process that the thread is part of, if a thread is selected. This option is available only if the process is running.
Process Properties	Lets you set debugger options temporarily for a particular process during the debugging session.
Dockable	Enables drag-and-dock for the **Thread Status** window.

3.2.17.2.15 Watch List Window

View ▶Debug Windows▶Watch List

Displays the current value of the watch expression based on the scope of the execution point. The **Watch List** window is a multi-tabbed view with each tab representing a distinct watch group. Only the watch group on the active tab is evaluated while debugging.

Tip: To enable or disable a watch expression quickly, use the check box ☑ next to the watch.

Item	Description
Watch Name	Shows the expression entered as a watch.
Value	Lists the current value of the expression entered.

Note: If the execution point moves to a location where any of the variables in an expression is undefined (out of scope), the entire watch expression becomes undefined. If the execution point reenters the scope of the expression, the Watch List

window displays the current value of the expression.

Tip: By grouping watches, you can prevent out of scope expressions from slowing down stepping.

Context menu

Right-click the **Watch List** window to display the following commands.

Item	Description
Edit Watch	Displays the **Watch Properties** dialog box that lets you modify the properties of a watch.
Add Watch	Displays the **Watch Properties** dialog box that lets you create a watch
Enable Watch	Enables a disabled watch expression.
Disable Watch	Disables a watch expression and so that it is not monitored as you step through or run your program. The watch settings remain defined. Disabling watches improves debugger performance.
Delete Watch	Removes a watch expression. This command is not reversible.
Copy Watch Value	Copies the text in the **Value** column of the selected watch to the clipboard.
Copy Watch Name	Copies the text in the **Watch Name** column of the selected watch to the clipboard.
Enable All Watches	Enables all disabled watch expressions.
Disable All Watches	Disables all enabled watch expressions.
Delete All Watches	Removes all watch expressions.
Add Group	Displays a dialog box, allowing you to name a watch group and add it to the watch list as a new tab.
Delete Group	Deletes a watch group from the watch list.

Move Watch to Group	Moves one or more selected watches to another watch group.
Stay On Top	Keeps the window visible when out of focus.
Show Column Headers	Toggles the display of the **Watch Name** and **Value** column titles.
Inspect	Displays information about the currently selected expression.
Dockable	Enables drag-and-dock for the **Watch List** window.

3.2.17.2.16 Disassembly

View ▶ Debug Windows ▶ CPU Windows ▶ Disassembly

Displays the address, the hexadecimal representation of the machine code instructions (opcodes), and the assembly instructions for each line of source code. The address is the offset into the disassembled method.

See Also

CPU Window (see page 1022)

3.2.17.2.17 Memory

View ▶ Debug Windows ▶ CPU Windows ▶ Memory

Displays the raw values contained in addressable areas of your program. Displayed only for unmanaged code. The pane displays the memory addresses, the current values in memory, and an ASCII representation of the values in memory.

There are four different views in order to view four distinct areas in memory at the same time. You can have more than one view and then use **Search ▶ Goto Address** in the separate views to look at different places in memory at the same time.

See Also

CPU Window (see page 1022)

3.2.17.2.18 Registers

View ▶ Debug Windows ▶ CPU Windows ▶ Registers

Displays the contents of the CPU registers of the 80386 and greater processors.

See Also

CPU Window (see page 1022)

3.2.17.2.19 Stack

View ▶ Debug Windows ▶ CPU Windows ▶ Stack

Displays the raw values contained in the program stack. Displayed only for unmanaged code.

See Also

CPU Window (see page 1022)

3.2.17.3 Code Explorer

View ▶Code Explorer

Navigates through the unit files. The Code Explorer contains a tree diagram that shows all of the types, classes, properties, methods, global variables, and global routines defined in your unit. It also shows the other units listed in the `uses` clause. Right-click an item in the Code Explorer to display its context menu.

When you select an item in the Code Explorer, the cursor moves to that item's implementation in the Code Editor. When you move the cursor in the Code Editor, the highlight moves to the appropriate item in the Code Explorer.

The Code Explorer uses the following icons:

Icon	Description
	Classes
	Interfaces
	Units
	Constants or variables (including fields)
	Methods or routines: Procedures (green)
	Methods or routines: Functions (yellow)
	Properties
	Types

Tip: To adjust the Code Explorer settings, choose Tools->Options->Delphi Options->Explorer

.

3.2.17.4 Customize Toolbars

View ▶Toolbars ▶Customize

Changes the toolbar configuration. Using this dialog box, you can add, remove, and rearrange buttons on toolbars.

Toolbars Page

The **Toolbars** page lists the toolbars you can show, hide, and reset.

Item	Description
Toolbars	Lists the toolbars available, such as Standard, Debug, and Desktop.
Reset	Returns any selected toolbar to its default configuration.

Commands Page

The **Commands** page displays the menu commands you can drag and drop onto a toolbar.

Item	Description
Categories	Lists the menus available, such as Debug and Run.
Commands	Lists all the commands available for the menu selected in the **Categories** list box. The icon to the left of the menu command is the button that will appear on the toolbar.

Options Page

The **Options** page allows you to specify that the IDE displays or hides tooltips and shortcut keys for toolbar button.

Item	Description
Show tooltips	Displays tooltips for toolbar buttons when you move the mouse over the button.
Show shortcut keys on tooltips	Displays any toolbar button shortcut keys in the tooltip text.

See Also

Customizing Toolbars (see page 156)

3.2.17.5 Data Explorer

View ▶ Data Explorer

Adds new connections, modifies; deletes, or renames your connections. You can browse database server-specific schema objects including tables, fields, stored procedure definitions, triggers, and indexes. Additionally, you can drag and drop data from a data source to a project to build your database application quickly. The **Data Explorer** commands available depend upon the object selected in the tree view. Commands are available for the following nodes:

- Provider types
- Provider connections
- Tables node
- Individual tables
- Individual views
- Individual stored procedures

Provider Types Commands

The following commands are available when you select nodes for providers types, such as DB2 and Interbase:

Item	Description
Refresh	Re-initializes all connections defined for the selected provider.
Add New Conection	Adds a new connection to the Data Explorer.
Migrate Data	Opens a a tabbed **Data Explorer** page for data migration in the **Code Editor**. This data migration page lets you select one or more tables from a source provider connection and a destination connection to which the tables will be migrated. Click Migrate to migrate the tables.

Individual Provider Commands

The following commands are available when you select nodes for individual provider connections:

Item	Description
Refresh	Re-initializes all connections defined for the selected provider.
Delete Connection	Deletes the current connection.
Modify Connection	Makes changes to the appropriate values in the editor.
Close Connection	Closes the current connection.
Rename Connection	Provides a new name to a named connection.

SQL Window	Opens the **Active Query Builder**, a tabbed page for writing and editing SQL statements in the **Code Editor**. This SQL window can be used to write, edit and execute SQL statements. When you execute the SQL, the results are displayed in the lower part of the page. For details on how to use the **Active Query Builder**, see http://www.activequerybuilder.com/hs15.html.

Tables Node Commands

The following commands are available when you select the Tables node for a connection:

Item	Description
Refresh	Re-initializes all connections defined for the selected provider.
New Table	Opens a tabbed **Data Explorer** page for table design in the **Code Editor**. This Table Design page can be used to specify the data structure for a new table. The Table Design page lets you add and remove columns, and alter column information. The Table Design page lets you change the following column information: Column Name, Data Type, Precision, Scale, and Nullable (that is, whether or not the column can be null). Right-click the page and choose Save Table to add the new table to your database.

Individual Table Commands

The following commands are available when you select individual tables:

Item	Description
Refresh	Re-initializes all connections defined for the selected provider.
Retrieve Data From Table	Opens a tabbed **Data Explorer** page in the **Code Editor**, displaying the data from the selected table. The Data Explorer page lets you sort and modify the data, but changes will not be saved back to the database.
Drop Table	Removes the selected table and all its data from the database.
Alter Table	Opens a tabbed **Data Explorer** page for table design in the **Code Editor**. This Table Design page can be used to modify the data structure for an existing table. The Table Design page lets you add and remove columns, and alter column information. The Table Design page lets you change the following column information: Column Name, Data Type, Precision, Scale, and Nullable (that is, whether or not the column can be null). If you make modifications, you are asked to save them when you close the Table Design page.
Copy Table	Copies the table structure and data for the selected table.
Paste Table	Migrates the table structure and data copied from a given provider to the selected provider. Although you must select a table in the target provider, no data will be overwritten.

View Commands

The following commands are available when you select individual views:

Item	Description
Refresh	Re-initializes all connections defined for the selected provider.
Retrieve Data From View	Opens a tabbed **Data Explorer** page in the **Code Editor**, displaying the data from the selected view. The Data Explorer page lets you sort and modify the data, but changes will not be saved back to the database.

Stored Procedure Commands

The following commands are available when you select individual stored procedures:

Item	Description
Locate Implementation	Navigate to the stored procedure's source code if you have a project with such source code open in the project group. Supported for Blackfish SQL only.
Refresh	Re-initializes all connections defined for the selected provider.
View Parameters	Opens a tabbed **Data Explorer** page for viewing and editing stored procedure parameter data in the **Code Editor**. The stored procedure can also be executed from this page.

See Also

Borland Data Providers for Microsoft .NET

Migrating Data Between Databases

ISQLSchemaCreate

BdpCopyTable

3.2.17.6 Delete Saved Desktop

View ▶Desktops▶Delete Desktop

Delete a saved desktop by selecting it from the list and clicking **Delete**.

See Also

Saving Desktop Layouts

3.2.17.7 Desktop Toolbar

Selects an existing desktop layout or saves the current settings as a desktop layout. The **Desktop** toolbar is located at the far right end of the upper toolbar in the IDE.

Item	Description
Dropdown list	Lists the distributed and user-defined desktop layouts. Click the desktop layout that you want to use, such as Default Layout, Classic Undocked, or Debug Layout.
Save current desktop	Displays the **Save Desktop** dialog box, allowing you to name and save the current desktop settings.
Set debug desktop	Sets the current desktop as the debug desktop, which is automatically displayed during runtime.

Tip: You can also choose View->Desktops

to manage your desktop settings.

See Also

Saving Desktop Layouts (↗ see page 161)

View Desktops Command (↗ see page 1059)

3.2.17.8 File Browser

View▶File Browser

Dockable Windows-style File Browser views files and directories and performs simple operations on files while you are running the IDE. The File Browser supports standard Windows context menu options, as well as the following commands that are specific to RAD Studio:

Item	Description
Open with RAD Studio.	Opens the selected file in the IDE.
Add to Project	Adds the selected file to the current project.

See Also

Using the File Browser (see page 166)

3.2.17.9 Add to Repository

Use this dialog box to add strings in the selected unit to the Translation Repository. This dialog is displayed when you right-click a node on the **Workspsace** tab of the Translation Manager and use the Add strings to repository comand.

To add individual strings, rather than adding the strings for an entire unit, right-click the string in the Translation Manager and choose **Repository** ▶ **Add strings to repository**.

The following options determine the criteria used for adding the strings.

Item	Description
Status	Adds strings based on the status displayed in the **Status** column on the **Workspace** tab. Check the statuses that you want to add.
Duplicate action	Determines how the repository responds when it finds a duplicate translation string for the same source string. **Skip** does not add the string. **Add** adds the string to the repository if no translated string exists for the original string. **Force Add** always adds the string to the repository, regardless of whether it exists in the repository. **Replace** overwrites the existing string with new string. **Display selection** offers the user a choice.
Include context information	Adds the unit path, and the value displayed in the **Id** column of the **Workspace** tab, to the Translation Repository. This context information is displayed in the status bar when you select a string in the Translation Repository.
Value	Indicates whether a string is added based on changes to its original value. **Changed** adds the string only if the original and translated values are different. **Unchanged** adds the string even if the original and translated values are the same. **Don't care** adds the string, whether it has changed or not, provided the string meets the other criteria set in this dialog box.
Comment	Adds or excludes strings based on the text in the **Comment** column on the **Workspace** tab. Type the comment text in the edit box and check **Include** to add strings with a matching comment, or check **Not include** to exclude strings with a matching comment.

Tip: To set general options for the Translation Repository, choose Tools->Translation Tools Options

and select **Repository**.

See Also

Localizing Applications (see page 18)

3.2.17.10 Message View

Displays messages such as compiler errors and warnings. You can copy one or more lines from the **Message** view to the clipboard.

The **Build** tab displays the build command. The **Output** tab displays build output messages. To select the verbosity level for build output, use the **Tools**▶**Options**▶**Environment Options** page.

3.2.17.11 Object Inspector

View ▶**Object Inspector**

Sets the properties and events for the currently selected object.

Tab	Description
Properties	Displays the properties for the selected object on a form.
Events	Displays the events for the selected object on a form.

Context Menu

Right-click the **Object Inspector** to display the following commands.

Item	Description
View	Filters the display of properties or events.
Arrange	Sorts the property or events by name or by category.
Revert to Inherited	Changes the property setting back to its original, inherited value.
Expand	Expands the selected property or event.
Collapse	Collapses the selected property or event.
Hide	Close the **Object Inspector**. To redisplay it, choose **View**▶**Object Inspector**.
Help	Displays this Help topic.
Properties	Displays the **Object Inspector Properties** dialog box, allowing you to change the appearance of the **Object Inspector**.
Stay on Top	Displays the **Object Inspector** on top of the desktop even if other windows are displayed.
Dockable	Enables drag-and-dock for the **Object Inspector**.

See Also

Setting Properties and Events (see page 164)

3.2.17.12 Project Manager

View▶**Project Manager**

Displays and organizes the contents of your current project group and any project it contains. You can perform several project management tasks, such as adding, removing, and compiling files.

Note: Some features described here are available only in specific editions of the product. For example, some functionality in the

Project Manager is available only for the C++ personality.

Item	Description
Project list box	Displays the projects in the current project group.
New	Displays the **New Items** dialog box so that you can add a new project to the current project group.
Remove	Removes the selected project from the current project group.
Activate	Displays the selected project on top of other projects in the IDE so that you can make changes to it. You can also double-click the project to activate it. The active project is displayed in bold.
Files	Displays a tree view of all the files in your project or project group. Click the plus sign (+) to display or the minus sign (-) to hide all the source files in your project.

Common Context Menu Commands

The **Project Manager** has different context menus, depending on what you select (file, project, project group, and so on). However, most context menus have the following common menu commands.

Item	Description
Auto Collapse	Collapses the tree structure of the project after you complete an operation.
Dockable	Docks (attaches) the **Project Manager** window to other tool windows, such as the **Code Editor**. Uncheck to make the **Program Manager** a floating window.
Show Path	Adds an additional field (**Path**) to the Project Manager to display the path of the files, projects, and project groups.
Status Bar	Displays the full path name of the selected file at the bottom of the **Project Manager** window.
Stay on Top	Displays the **Project Manager** on top of the desktop even if other windows are displayed.
Toolbar	Shows or hides the toolbar on the top of the **Project Manager**.

Project Group Context Menu

Right-click a project group to display the following commands.

Item	Description
Add New Project	Displays the **New Items** dialog box, allowing you to create a new project and add it to the current project group.
Add Existing Project	Displays the **Open Project** dialog box, allowing you to add an existing project to the current project group.
Customize...	Opens the Customize New menu, which allows you to select items, including menu-item separators, from a gallery and add them to your Add New... menu on the Project Manager.
Save Project Group	Saves the project file (.bdsgroup) for the project group. Use this command after adding, removing, or changing the order of projects in a project group.
Save Project Group As	Displays the **Save As** dialog box, allowing to save the project with a new name and location.
Rename	Renames the project file.
Configuration manager	Opens the Build Configuration Manager dialog box. Use this dialog box to apply the active configuration to a project or several projects.

Project Context Menu

Right-click a project file to display the context menu commands.

Item	**Description**				
Add	Displays the **Add to Project** dialog box, allowing you to add files to the selected project.				
Add New Unit	Adds a compilation unit to the selected project, assigning the name `Unit01.cpp` for the first unit added, and then augmenting the number for each subsequent unit added. To rename a unit and all its components, right-click the unit name in the Project Manager, and select Rename.				
Add New Form	Adds a new form to the selected project and displays the new form in the Code Editor.				
Add New Password Dialog	Adds a `PassWord.cpp` node to the selected project in C++Builder, including a PassWord.dfm file, and sets up a template for creating a PassWord unit in your project.				
Add New Other...	Displays the **New Items** dialog box and lists the item categories that are available to you, such as C++Builder Files. Click a category in the left-hand pane, and then the right-hand pane displays the items of that category that you can add to your project.				
Add New > Directory View	Displays the **Directory View** or **Folder** dialog box, which allows you to select a directory to add to the **Project Manager**. Adding a Directory View adds a yellow-colored folder node to the tree structure.				

Add New > Virtual Folder	(C++ only) Adds a greyed-out virtual folder to the selected project. Virtual folders are displayed by default last in the tree structure. Use the context menu for the virtual folder to manage the folder.				
Add New > Customize...	Displays the **Customize New Menu** dialog box and allows you to customize the **File ▶ New** menu.				
Add Reference	Displays the **Add Reference** dialog box, allowing you to add an assembly, COM type library, or project reference to the project.				
Add Web Reference	Displays the **Add Reference** dialog box, allowing you to Web reference to your client application and access the Web Service you want to use.				
Browse with...	Displays the **Browse With** dialog box where you can choose an external browser to use for viewing HTML-based files. You can also add, remove, or edit browsers in browser list in this dialog box.				
Show Markup	Displays the source view for HTML-based files in the editor.				
Show Designer	Displays an HTML-based file in design view.				
Remove File	Displays the **Remove From Project** dialog box, allowing you to remove a file from the selected project.				

Save	Stores changes made to all files in the open project using each file's current name. If you try to save a project that has an unsaved code file, the product opens the **Save As** dialog box, where you can enter the new filename.				
Save As...	Displays the**Save As** dialog box for each compilation unit that needs to be saved and allows you to enter a new name for each unit.				
Rename	Highlights the project node and allows you to enter a new name or edit the existing name of the project.				
Remove Project	Removes the active project from its project group. Removing the target from the current project group affects the project group's project file (`.bdsgroup`); it does not remove any files from disk. Thus, remove a project from your project group before deleting its file from disk so that the product can update the project file accordingly.				
Activate	Makes the current project active.				
Clean	Removes generated files from the project, such as object code files.				
Make	Compiles all files in the current project that have changed since it was last built and any files that depend on them. You can also choose **Project ▶ Compile [project name]**.				

Build	Rebuilds all files in your project regardless of whether they have changed. You can also choose **Project▶Build [project name]**.			
Close	Closes all the open files of the active project. Before closing the file, the IDE prompts you to save any changes. If you have not previously saved the project, or any file, the product opens the **Save As** dialog box, where you can enter the new filename.			
Build Sooner	Moves a project up in the list of projects within a project group, which changes the order in which the projects are compiled.			
Build Later	Moves a project down in the list of projects within a project group.			
Make All From Here	Compiles only the selected projects and all others listed below if they have changed since the last build. In contrast, choose **Project▶Compile All Projects** to rebuild every project in the project group that has changed.			
Build All From Here	Rebuilds only the selected project and all others listed below it regardless of whether they have changed. In contrast, choose **Project▶Build All Projects** to rebuild all projects in the project group.			
Dependencies	Displays the **Project▶Dependencies** dialog box, allowing you to change the build order for the projects in a project group.			

Options	Displays the **Project**▶**Options** dialog box, allowing you to change the selected project's application and compiler options.			
Set as Start Page	Specifies the selected HTML-based web form or web service as the Start Page for an ASP project. The Start Page is the page that is displayed in the browser when the project is run from within the IDE. The Set as Start Page option is enabled ro `.aspx` files, `.asmx` files, web forms, and web services.			
View in Browser	Displays the elected HTML-based file in an external browser.			

Show All Files	Shows all the project's files in the **Project Manager**. This menu item is a toggle, so you can click it to hide files.	ASP.NET Configuration	Runs the Microsoft ASP.NET Web Site Administration tool. This provides a UI for editing the **web.config** file for ASP.NET projects. See http://msdn2.microsoft.com/en-us/library/yy40ytx0(vs.80) for more information.	View Source	Displays the source code for the selected project.

Deployment Context-Menu

Right-click the Deployment node in the Project Manager to display the context menu commands. The commands vary depending on the type of project being deployed.

Item	Description
New ASP.NET Deployment	Opens the Deployment Manager for an ASP.NET project.

File Context Menu

Right-click a file within a project to display the context menu commands. The commands vary depending on the type of file selected.

Item	Description
Add Reference	Displays the **Add Reference** dialog box, allowing you to add an assembly, COM type library, or project reference to the project.
Add Web Reference	Displays the **Add Reference** dialog box, allowing you to Web reference to your client application and access the Web Service you want to use.
Build	Is a shortcut that compiles the selected file.
Copy Local	Copies the assembly to the local output directory. By default, Copy Local is checked for assemblies that are not in the Global Assembly Cache (GAC).
Edit Local Options	For C++, displays an abbreviated **Project Options** dialog box that contains only the Paths and Defines page, nine pages of C++ compiler options, and the Build Events page.
Open	Opens the selected files in the **Code Editor**.
Preprocess	Runs the C++ preprocessor (`cpp32`).
Remove From Project	Removes the selected files from the project. You will be prompted to save any changes.
Save	Saves changes made to the selected files using their current names.
Save As	Displays the **Save As** dialog box, allowing you to saves the selected files with new names and locations.
Show Dependencies	Displays
Rename	Allows you to rename the file and any corresponding secondary files that appear as child nodes in the **Project Manager**.

Build Configurations Context Menu (C++)

Right-click either the Build Configurations node in the Project Manager or the name of a specific build configuration within the node to display the context menu.

Item	Description
Add New	Adds a child configuration, based on the selected configuration, and listed in the **Project Manager** under the name of the parent configuration..
Save As	Displays the **Save as** dialog box and allows you both to save the selected configuration to a specific location and to rename the saved file.
Rename	Allows you to rename the selected build configuration.
Delete	Displays the **Confirm** dialog box and allows you to delete the selected configuration.
Activate	Makes the selected build configuration the current active configuration for the project. The active build configuration is listed in boldface.
Apply Option Set	Displays the Apply Option Set dialog box and allows you to select an .optset file to apply to the selected build configuration. You can choose to overwrite, replace, or preserve the existing option values.

Edit	Displays the **Project**▶**Options** dialog box preloaded with the values set in the selected build configuration.

3.2.17.13 Save Desktop

View ▶**Desktops**▶**Save Desktop**

Saves your current IDE desktop arrangement as a desktop layout.

Item	Description
Save current desktop as	Enter a new name for the desktop or select a name from the drop-down list.

See Also

Saving Desktop Layouts

3.2.17.14 Select Debug Desktop

View ▶**Desktops**▶**Set Debug Desktop**

Determines which saved desktop layout is used when you are debugging.

Item	Description
Debug desktop	Select a desktop layout from the drop-down list.

See Also

Saving Desktop Layouts

3.2.17.15 Structure View

View▶**Structure**

Shows the hierarchy of source code or HTML displayed in the **Code Editor**, or components displayed on the **Designer**. When displaying the structure of source code or HTML, you can double-click an item to jump to its declaration or location in the **Code Editor**. When displaying components, you can double-click a component to select it on the form.

If your code contains syntax errors, they are displayed in the **Errors** node in the **Structure View**. You can double-click an error to locate the corresponding source in the **Code Editor**. ((Not applicable for C++ development.)

Tip: You can control the content and appearance of the Structure View

by choosing **Tools**▶**Options**▶**Environment Options**▶**Explorer** and changing the settings.

Context Menu

Right-click the **Structure View** to display the following commands. The commands on the context menu depend on whether source code or components are displayed in the **Structure View**.

Item	Description
New	Adds a new node to the **Structure View**.

Rename	Changes the name of the selected node in the **Structure View**.
Edit	Displays a submenu allowing you to undo changes, cut, copy, paste, delete, or select all of the controls on the **Designer**.
Control	Displays a submenu allowing you to bring the selected control to the front or send it to the back of the the **Designer**.
Properties	Displays the **Explorer Options** dialog box allowing you to change the content and appearance of the **Structure View**.
Stay on Top	Displays the **Structure View** on top of the desktop even if other windows are displayed.
Dockable	Enables drag-and-dock for the **Structure View**.

Toolbar (C++)

The **Structure** view contains a toolbar for C++ application development that allows you to control how the contents of the **Structure** view are displayed. It consists of the following buttons:

Sort Alphabetically	Sorts the contents of the **Structure** view alphabetically.
Group by Type	Groups items into folders by **type** in the **Structure** view.
Group by Visibility	Groups class members into folders by visibility: **public**, **protected**, **private**, and **published**. For C++ , 'Classes' is a generic group that encompasses **classes**, **structs**, **unions** and **templates**.
Show Type	Displays the **type** to the right of the member in the **Structure** view.
Show Visibility	Toggles the **Structure** view display through different visibility levels: Show **public** only, show **public** and **protected**, show **public**, **protected**, and **private**, and show all.

3.2.17.16 Templates Window

View ▶ Templates

Creates, edits, or deletes live code templates.

Item	Description
Name	Displays the name of the available live templates.
Description	Describes the live template.

Templates Window Toolbar

Toolbar Button	Description
New Code Template	Creates an XML template file in the code editor with default code which you can edit. Specify the template name, author, description, and template content.
Remove Code Template	Removes the selected template from the list and deletes the template's .xml file from disk.
Edit Code Template	Opens the content of the selected template file in the code editor where you can modify it.
Insert Live Template into Code Editor	Inserts the code content from the selected template into the code editor at the cursor location.
Filter Code Templates by Language	Displays only the templates related to the current project language.

See Also

Creating Live Templates (see page 138)

3.2.17.17 To-Do List

View ▶To-Do List

Creates and manages a to-do list.

Item	Description
Action Item	This column includes a check box, an icon, and the task. The check box indicates whether the item has been completed. A window icon indicates the item was entered in the to-do list. A unit icon indicates it is a comment in the source code. The text is the actual task to be done. Grayed text indicates the item comes from a source file that is part of the current project but is not open in the **Code Editor**. Bold text indicates the source is open in the **Code Editor**. Double-click an item to open its source in the editor.
Priority	Specifies the importance of the item using a decimal number from 1 (the highest) to 5 (the lowest).
Module	For items that have been added as code comments, indicates the path and module in which the comment exists.
Owner	Indicates who is responsible for completing the task.
Category	Indicates a type of task, for example, user interface task or an error handling task.

Tip: To sort the list, click any column heading. For additional processing options, right-click and use the context menu commands.

See Also

Using To-Do Lists (see page 166)

3.2.17.18 Add or Edit To-Do Item

Adds items to a to-do list or to change an item.

Item	Description
Text	Specifies the to-do list item text.
Priority	Specifies the importance of the item using a decimal number from 1 to 5. Type the number or select one using the spin control.
Owner	Indicates who is responsible for completing the task. Type the name or select one if others are listed in the spin control.
Category	Indicates the type of task, for example, a user interface or UI, or Interface implementation. Type the category or select one if others are listed in the spin control.

See Also

Using To-Do Lists (see page 166)

3.2.17.19 Filter To-Do List

Controls which items are displayed in a to-do list.

Item	Description
Filter by	Uncheck a category, owner, or item type to hide it from the to-do list.
Show All	Checks everything in the **Filter by** list.

See Also

Using To-Do Lists (see page 166)

3.2.17.20 Table Properties

Controls the appearance of the resulting to-do list when using the **Copy as▸HTML table** command from the right-click menu of the **To-do List** dialog.

Table tab

Specifies the properties for the HTML table used to display the to-do list.

Item	Description
Caption	Specifies a caption for the table.
Border Width	Specifies the width (in pixels only) of the frame around a table.
Width (Percent)	Specifies a value for how wide the table will appear on the page. The value is relative to the amount of available horizontal space.
Cell Spacing	Specifies how much space to leave between the left side of the table and the left side of the leftmost column, the top of the table and the top-side attribute. Also specifies the amount of space to leave between cells.
Cell Padding	Specifies the amount of space between the border of the cell and its contents.
Background Color	Indicates a background color for the HTML table cells.
Alignment	Indicates the location (left, right, or center) of the table on the HTML page.

Columns tab

Specifies the properties for each of the columns in the to-do list.

Item	Description
Column	Indicates the column for which you want to specify properties.
Alignment	Specifies the alignment of the text within the column (left, right, or center).
Vertical alignment	Specifies the alignment of the text within the cell (top, middle, or bottom).
Title	Indicates the column heading.
Width	Specifies the width of the column in a percentage of the whole table width.
Height	Specifies a recommended cell height in pixels.
Wrap text	Allows text within cells to wrap.
Visible	Determines whether or not this column will be included in the table or not.

Font Size	Specifies the point size of the text in the column.
Face	Sets the typeface of the text in the column.
Color	Sets the color of the column.
Bold	Makes the text in the column bold.
Italic	Makes the text in the column italic.

See Also

Using To-Do Lists (see page 166)

3.2.17.21 Tool Palette

View ▶Tool Palette

Assists with a new project, adds components to a form, or adds code snippets to the **Code Editor**.

Item	Description
Categories	Displays a list of the item categories, allowing you to position the **Tool Palette** to a category.
	Sets or removes the **Tool Palette** filter. Click anywhere in **Tool Palette** and begin typing the name of the item you want to locate. The **Tool Palette** is automatically displays only those items that match what you type. Click the filter icon to remove the filter.

Tip: To reorder categories or items in the Tool Palette

, click the item or category and drag and drop it elsewhere on the **Tool Palette**. The context menu command Lock Reordering disables/enables drag and drop reordering.

Context Menu

Right-click the **Tool Palette** to display the following commands.

Item	Description
Add New Category	Displays the **Create a New Category** dialog box, allowing you to create an empty category. You can then drag-and-drop components from other categories into the new category to create a customized category.
Delete Category	Deletes the selected category from the **Tool Palette**. To restore the category, choose Customize .NET Components.
Delete Button	Deletes the selected item from the **Tool Palette**.
Hide Button	Removes the selected item from the **Tool Palette**, but does not delete it.
Unhide Button	Redisplays items previously hidden by using the Hide Button.
Installed .NET Components	Displays the **Installed .NET Components** dialog box, allowing you to add components to the **Tool Palette**. This command is available only when the **Tool Palette** contains components.
Clear All Code Snippets	Removes user defined code snippets from the **Tool Palette**.
Auto Collapse Categories	Allows only one category to be expanded at a time.
Collapse All	Displays only the categories of items.
Expand All	Displays the items in each category.

Lock Ordering	Disables drag and drop reordering of categories and items on the **Tool Palette**.
Reset Palette	Removes all **Tool Palette** customizations.
Properties	Displays the **Tool Palette** page of the **Options** dialog box, allowing you to change the appearance of the **Tool Palette**.
Stay on Top	Displays the **Tool Palette** on top of the desktop even if other windows are displayed.
Dockable	Enables drag-and-dock for the **Tool Palette**.

See Also

Adding Components to the Tool Palette (↗ see page 160)

Adding Components to a Form (↗ see page 152)

Using Code Snippets (↗ see page 148)

Finding Items on the Tool Palettes (↗ see page 158)

3.2.17.22 Translation Manager

View ▶Translation Manager

Views and edits language resource files.

Item	Description
Project tab	Displays the following tabs: The **Languages** tab lists each language in the open project and the locale ID, file extension, and translation directory for each language. The **Files** tab lists the resource files for the language selected on the **Languages** tab. Double-click a resource file to open it in a text editor.
Workspace tab	Displays a tree view of the project. When you select a non-resource in the left pane, summary information appears in the right pane. When you select a resource file in the left pane, a grid for viewing and editing translations appears.

Workspace Tab Actions Context Menu

The Actions context menu provides quick access to commonly used function while using the Translation Manager/External Translation Manager. To access the Actions menu, in the Workspace tab you can either double-click the **Actions** button or right-click the grid.

Item	Description
Filters	Lets you can display or hide rows based on the following criteria: Show All, Toggle (jumps between displaying criteria that are either checked and unchecked in the Columns context menu), Show None, Show Untranslated, Show Translated, Show Newly Translated, Show Auto Translated, and Show Unused (translations that were kept even after the resource was later deleted).
Columns	Lets you display or hide columns by checking the column name. Show All displays all columns, Toggle jumps between displaying criteria that are either checked and unchecked in the Columns context menu), and Show None removes all of the columns from the display.
Edit	Displays the **Edit Selection** dialog box where you can edit the value for the target language, status, or comment field.

Repository	Displays the following commands: Add strings to Repository stores the translation from the selected row(s) into the Repository database. Get strings from Repository searches the Repository for a translation in the target language whose source string matches the selected resource.
Font	Displays the **Font** dialog box, allowing you to change the font of the values in the base language column, the target language column, and the comment column.
Copy previous line	Overwrites the selected value in the target language column with the value from the cell immediately above (only if the value is the same type, such as a number or a text string).
Copy previous translation	Pastes an earlier version of the source strings and their translations that have been overwritten by the updater.
Copy from original	Overwrites the target language value with the base language value and changes the status from **Translated** to **Untranslated**.
Change status to Translated	Changes **Untranslated** to **Translated** in the **Status** column. You can also click the arrow in each **Untranslated/Translated** drop-down box to change the status.
Find next untranslated item	Jumps to the next row in the grid that has a status of **Untranslated**.
Select all	Selects the values in all rows and columns.

Workspace Tab Keyboard Shortcuts

The following keyboard shortcuts are available in the **Workspace** tab:

Item	Description
Ctrl+A	Select everything in grid.
Ctrl+C	Copy selection to clipboard.
Ctrl+D	Copy translation from prior row in grid.
Ctrl+E	Show Multiline editor.
Ctrl+F	Show Find dialog box.
Ctrl+K	Keep forms on top.
Ctrl+N	Find next untranslated item.
Ctrl+O	Copy text from source (base language) column into translation column.
Ctrl+P	Restore translation from Previous translation column.
Ctrl+Q	Show Actions context menu.
Ctrl+S	Save translations.
Ctrl+T	Change status to Translated.
Ctrl+V	Paste selection from clipboard.
Ctrl+W	Reset column width.
Ctrl+X	Cut selection and add to clipboard.
F6	Switch between left and right panes in Workspace tab.
Ctrl+F5	Refresh translated form.
Ctrl+F7	Show original form.
Ctrl+F8	Show translated form.
Shift+Ctrl+F5	Refresh grid.

Shift+Ctrl+F7	Synchronize the translated version with the base language version, for example, when changes are made to the base version.

See Also

Localizing Applications (see page 18)

Adding Languages to a Project (see page 169)

Editing Resource Files in the Translation Manager (see page 170)

3.2.17.23 Multi-line Editor

Edits translations that are lengthy or contain multiple lines of text separated by hard returns. The editor displays the source and target languages in separate panes. Only the target language is editable.

Item	Description
Up and Down arrows	Moves to the previous or next string in the Translation Manager grid. Clicking either button saves your changes.
Tile Across and Tile Down	Changes the orientation of the editor panes.
Save	Saves your changes.
Close	Discards unsaved changes and close the editor.
Font	Changes the display font in the editor. The changes apply to whichever pane the cursor is in when you click the **Font** button and affect the Translation Manager grid as well as the Multi-line editor.
Word Wrap	Enables or disables wrapping of long lines.

See Also

Localizing Applications (see page 18)

Adding Languages to a Project (see page 169)

Editing Resource Files in the Translation Manager (see page 170)

3.2.17.24 Type Library Editor

View ▶Type Library

Makes changes to your type library. The Type Library editor generates the required IDL syntax automatically. Any changes you make in the editor are reflected in the corresponding implementation class (if it was created using a wizard).

The **View▶Type Library** command is available only for projects that contain a type library. The wizards on the ActiveX page automatically add a type library to the project when they create a COM object.

Object List Pane

Each instance of an information type in the current type library appears in the object list, represented by a unique icon. Select an icon to see its data pages displayed in the information pane at the right.

Attributes Page

Lists the type information associated with the object currently selected in the object list pane. You can use the controls to edit these values. What attributes appear depends on the selected element.

Item	Description
Name	A descriptive name for the type library. The name can’t include spaces or punctuation.
GUID	The globally unique 128-bit identifier of the type library’s interface (a descendant of ITypeLib).
Version	A particular version of the library in cases where multiple versions of the library exist. The version is either a pair of decimal integers separated by a period, or a single decimal integer. The first of the two integers represents the major version number, and the second represents the minor version number. If a single integer is used, it represents the major version number. Both major and minor version numbers are short unsigned integers in the range between 0 and 65535, inclusive
LCID	The locale identifier that describes the single national language used for all text strings in the type library and its elements.
Help String	A short description of the type library. Used with Help Context to provide Help as a Help file. This string is mapped to the Help Context when creating the help file.
Help Context	The Help context ID of for the type library’s main help. This ID identifies the Help topic within the Help file.
Help String Context	For help DLLs, the Help context ID of the type library’s main help. Used with Help String DLL to provide Help as a separate DLL.
Help String DLL	The fully-qualified name of the DLL used for help, if any.
Help File	The name of the help file (.hlp) associated with the type library, if any.

Note: The Type Library editor supports two mechanisms for supplying help. The traditional help mechanism, where a standard windows help file has been created for the library, or where the help information is located in a separate DLL (for localization purposes). You must supply the help file to which the Help attributes apply.

Text Page

Contains the declarations for the currently selected element in IDL or Object Pascal. You can use this page to enter changes more quickly than using the other pages or to review all the type information at once.

All type library elements have a text page that displays the IDL or Object Pascal syntax for the element. The Type Library page of the Environment Options dialog determines which language is used on the text page. Any changes you make in other pages of the element are reflected here. If you add IDL or Object Pascal code directly in the text page, changes are reflected in the other pages of the Type Library editor.

Note: The Type Library editor generates syntax errors if you add IDL identifiers that are currently not supported by the editor; the editor currently supports only those IDL identifiers that relate to type library support (not RPC support).

Flags Page

Lists various attributes that modify the object described on the Attributes page. This page is not available for all elements.

Some type library elements have flags that let you enable or disable certain characteristics or implied capabilities. The flags page lists several check boxes let you turn these flags on or off.

Uses Page

Only available when the type library is selected in the **Object List** pane. Lists other type libraries that contain definitions on which this one depends.

To add a dependency, check the box to the left of the type library name. The definitions in that type library can then be used by this one. If the type library you want to add is not in the list, right click and choose Show All Type Libraries.

To remove a dependency, uncheck the box to the left of a type library name.

To view one of the other type libraries, select that type library, right click and choose View Type Library.

Implements Page

Only available when a CoClass is selected in the **Object List** pane. Lists the interfaces that the CoClass implements. Use this page to change the interfaces associated with the object or change their properties.

Item	Description
Interface	Name of an interface or dispinterface that the CoClass supports. Note that the name for interfaces and dispinterfaces is assigned on the Attributes page when the interface is selected.
GUID	The globally unique identifier for the interface. This column is informational only: its value can't be changed.
Source	Indicates whether the interface functions as an event source. If so, the CoClass does not implement the interface. Rather, clients implement the interface and the CoClass calls them using this interface when it fires events.
Default	Indicates that the interface or dispinterface represents the default interface. This is the interface that is returned by default when an instance of the class is created. A CoClass can have two default members at most. One represents the primary interface, and the other represents an optional dispinterface that serves as an event source.
Restricted	Prevents the item from being used by a programmer. An interface cannot have both restricted and default attributes.
VTable	Indicates whether interface methods can be called using a VTable (as opposed to IDispatch calls). This column is informational only: its value can't be changed.

COM+ Page

Use this page to change the transaction attribute of a transactional object you will install with MTS or the COM+ attributes of a CoClass you will install with COM+.

You can also use this page for Automation objects that were not created using the **Transactional Object** wizard, and it will influence the way the IDE installs them into MTS packages or COM+ applications. However, objects that are not created using the wizard do not automatically include support for IObjectControl. This means that they are not notified about activation and deactivation (and so do not have OnActivate and OnDeactivate events). They also do not have an ObjectContext property. You must therefore obtain the object context by calling the global GetObjectContext function.

Note: Only the Transaction Model attribute is used when installing into an MTS package, all other settings are ignored. If you intend to install the object under MTS, it must be an Automation object in an in-process server (DLL).

Warning: The attributes you specify on the COM+ page are encoded as custom data in the type library. This data is not recognized outside of Delphi. Therefore, it only has an effect if you install the transactional object from the IDE. If you deploy your object in any other way, these settings must be explicitly set using the MTS Explorer or COM+ Component Manager.

Item	Description
Call Syncronization	COM+ only: Determines how the object participates in activities. These provide additional synchronization support beyond that supplied by the threading model.
Transaction Model	Specifies the transaction attribute, which indicates how the object participates in transactions, if at all. The possible values differ depending on whether the object is to be deployed under MTS or COM+. Note that if the transaction attribute indicates that transactions are supported, Just In time Activation must be enabled.
Object Pooling	COM+ only: Determines whether object instances can be pooled. When enabling Object Pooling, it is your responsibility to ensure that the object is stateless.

Creation TimeOut	COM+ only: Determines how long, in milliseconds, a pooled object remains in the object pool before it is freed.
Allow Inproc Subscribers	Only applicable when the CoClass represents a COM+ event object. Determines whether in-process applications can register interest as clients of the event object.
Fire In Parallel	Only applicable when the CoClass represents a COM+ event object.Determines whether COM+ fires events in parallel (on multiple threads), or one by one on the same thread.

Parameters Page

Only available when a property or method is selected in the **Object List** pane. It lets you set the parameters and return value for methods (including property access methods).

Item	Description
Name	Represents the parameter name. You can edit the value directly.
Type	Represents the data type of the parameter. Select an available type from the drop-down list that appears when you click in the Type column.
Default Value	Specify a default value for an optional parameter by typing it into the column. All subsequent parameters must be optional. Any preceding optional parameters should also have a default value. When working in Object Pascal, local IDs are specified using a parameter type specifier of TLCID. In IDL, this is specified using a parameter modifier.

3.2.17.25 View Form

View ▶Forms

Views any form in the current project. When you select a form, it becomes the active form, and its associated unit becomes the active module in the **Code Editor**.

3.2.17.26 View Units

View ▶Units

Views the project file or any unit in the current project. When you open a unit, it becomes the active page in the **Code Editor**.

3.2.17.27 Window List

View ▶Window List

Displays a list of open windows.

Item	Description
Windows	Select a window and click **OK** to display that window.

3.2.17.28 New Edit Window

View ▶New Edit Window

Brings up a new **Code Editor** window as a separate window. The previous **Code Editor** window remains open.

3.2.17.29 Toggle Form/Unit

View ▶ Toggle Form/Unit

Toggles the view between Form and Unit.

See Also

Form Designer (↗ see page 46)

Program and Units (↗ see page 683)

3.2.17.30 Model View Window

View ▶ Model View

Shows the logical structure and containment hierarchy of your project. Note the ECO framework is available only for C# and Delphi for .NET, and in the Architect SKU and higher. ECO-related icons and topic links are unavailable in other product SKUs.

Code Visualization Icon	Represents
	A project
🗀	A UML package (ECO framework)
🗎	A UML package (code visualization)
▤	A class (ECO framework)
▤	A class (code visualization)
▭	An interface (code visualization)
▤	An operation (ECO framework)
◈	An operation (code visualization)
▤	A property (ECO framework)
◈	A property of a class (code visualization)
	The diagram for the project or UML package
↗	A link to another class or interface (code visualization)
⇗	Generalization (ECO framework)
⊢⊣	Association (ECO framework)
▤	Derived association (ECO framework)

Note: On code visualization diagrams, each .NET namespace declaration corresponds to a UML package (this is *not* true for ECO-enabled source code). Double-clicking on a namespace node in the Model View tree

cannot open a specific source code file, since namespaces can span multiple source files.

Tip: To quickly open the source code editor

for a specific class, interface, or member, double-click the item in the **Model View tree**.

See Also

Using Code Visualization

Using the Model View Window and Code Visualization Diagram

Using the Overview Window

UML Features in Delphi for .NET

Overview of the ECO Framework

Integrated Modeling Tools Overview

Using the ECO Wizards

Importing a Model

Using the ECO Space Designer

Using the OCL Expression Editor

Building an ECO Enabled User Interface

Deploying an ECO Application

3.2.17.31 CodeGuard Log

View ▶ Debug Windows ▶ CodeGuard Log

Provides runtime debugging for C++ applications being developed. CodeGuard reports errors that are not caught by the compiler because they do not violate the syntax rules. CodeGuard tracks runtime libraries with full support for multithreaded applications.

See Also

Errors reported by CodeGuard

Warnings reported by CodeGuard

Using CodeGuard

3.2.17.32 Desktops

View ▶ Desktops

Allows you to choose between preset desktop layouts. Desktop layouts can be used to create and manage windows.

Item	Description
None	Does not specify a preset desktop layout.
Classic Undocked	Emulates earlier Delphi versions, with separate windows for the menus and palette, designer, etc.
Debug Layout	Customized for debugging, with call stack, thread, and other views shown instead of the default windows used for designing applications.
Default Layout	Shows all windows docked into one container, with the most-used designing windows shown, including the tool palette, object inspector, design form, etc.

See Also

Saving Desktop Layouts (↗ see page 161)

Desktop Toolbar (see page 1036)

3.2.17.33 Dock Edit Window

View ▶Dock Edit Window

Sizes new **Code Editor** windows to fit appropriately inside the IDE. You can reselect Dock Edit Window to toggle between the new **Code Editor** window and the original **Code Editor** window.

See Also

New Edit Window (see page 1057)

3.2.17.34 Find Reference Results

View ▶Find Reference Results

Brings up the **Find References** pane. This pane is dockable and is used in conjunction with the **Search▶Find** function.

See Also

Finding References (see page 141)

Finding References Overview (Delphi (see page 65)

Find References (see page 954)

3.2.17.35 Help Insight

View ▶Help Insight

Displays a hint containing information about the symbol such as type, file, location of declaration, and any XML documentation associated with the symbol (if available).

Alternative ways to invoke Help Insight is to hover the mouse over an identifier in your code while working in the **Code Editor**, or by pressing `CTRL+SHIFT+H`.

See Also

Code Editor (see page 42)

3.2.17.36 Show Borders

View▶Show Borders

Displays gray borders that represent page margins in the Diagram View and Overview. Diagrams exist within the context of a namespace (or a package). This feature is only applies to ASP .NET applications.

See Also

Together Diagram Overview (see page 90)

Creating a Diagram (see page 196)

Together Diagram Appearance Options (see page 1089)

3.2.17.37 Show Grid

View ▶ Show Grid

Shows the design grid in the background behind diagrams. Diagrams exist within the context of a namespace (or a package). This feature is only applies to ASP .NET applications.

To turn on the grid, open **Tools ▶ Options ▶ HTML/ASP.NET**. In the **Designer Options** select Grid Layout from the pull-down menu.

See Also

Together Diagram Overview (see page 90)

Creating a Diagram (see page 196)

Together Diagram Appearance Options (see page 1089)

3.2.17.38 Show Tag Glyphs

View ▶ Show Tag Glyphs

Displays tags in an ASP form. This feature only applies to ASP .NET applications.

3.2.17.39 Snap To Grid

View ▶ Snap To Grid

Allows diagram elements to "snap" to the border of a control to the nearest coordinate of the diagram background design grid. The snap function works whether the grid is visible or not. Diagrams exist within the context of a namespace (or a package). This feature is only applies to ASP .NET applications.

See Also

Together Diagram Overview (see page 90)

Creating a Diagram (see page 196)

Together Diagram Appearance Options (see page 1089)

3.2.17.40 Toolbars

View ▶ Toolbars

Allows you to choose the toolbars that are displayed in the IDE.

Item	Description
Standard	Adds buttons that are used for opening and saving files and other common tasks.
Debug	Adds buttons that are used for stepping, tracing, and other debugging tasks.
Custom	Adds buttons that links to Help Contents. See link below.
Spacing	Adds buttons that control spacing of components on the design form.
Position	Adds buttons that control positioning and visibility of components on the design form.

Personality	Adds buttons that display the current personality.
Browser	Adds buttons that allow browser-style navigation of code.
HTML Design	Adds buttons for HTML document elements.
HTML Format	Adds buttons that format HTML text.
HTML Table	Adds buttons that insert and position HTML tables.
View	Adds the Unit, Form and Toggle between the two buttons.

See Also

Packages (↗ see page 156)

3.2.17.41 Translation Editor

View ▶ Translation Manager ▶ Translation Editor

Edits resource strings directly, adds translated strings to the Translation Repository, or gets strings from the Translation Repository.

See Also

Editing Resource Files in the Translation Manager (↗ see page 170)

Translation Manager (↗ see page 1052)

3.2.17.42 Welcome Page

View ▶ Welcome Page

Opens the product's Welcome Page, which displays lists of your recent projects and favorites. The Welcome Page also contains links to developer resources, such as product-related online help. As you develop projects, you can quickly access them from the list of recent projects at the top of the Welcome Page.

See Also

IDE Tour (↗ see page 34)

3.2.18 Win View

Topics

Name	Description
Assembly Metadata Explorer (Reflection viewer) (↗ see page 1063)	**File ▶ Open...** Use the assembly metadata explorer (Reflection viewer) to inspect types contained within a .NET assembly.
Type Library Explorer (↗ see page 1065)	**File ▶ Open...** Use the type library explorer to inspect types and interfaces defined within a Windows type library.
Search (↗ see page 1066)	Use this dialog box to search for various kinds of elements (such as classes or interfaces), or for a specific element within a .NET assembly, or a type library.

3.2.18.1 Assembly Metadata Explorer (Reflection viewer)

File ▶ Open...

Use the assembly metadata explorer (Reflection viewer) to inspect types contained within a .NET assembly.

Icon	Type	Available Tabs
clr	Assembly	**Properties**, **Attributes**, **Flags**, **Uses**
📁	Namespace	**Properties**
●	Class	**Properties**, **Attributes**, **Flags**, **Implements**
●	Sealed Class	**Properties**, **Attributes**, **Flags**, **Implements**
🔑	Interface	**Properties**, **Attributes**, **Flags**, **Implements**, **Implementors**
➡	Method	**Properties**, **Attributes**, **Flags**, **Parameters**, **Call Graph**
⇄	Method with return value	**Properties**, **Attributes**, **Flags**, **Parameters**, **Call Graph**
◈	Property with getter and setter	**Properties**, **Flags**
◈	Property Getter Method	**Properties**, **Flags**
◈	Property Setter Method	**Properties**, **Flags**
◇	Field	**Properties**, **Flags**
⚡	Event	**Properties**, **Attributes**, **Flags**

The metadata fields shown on each tab differ according to the type of item selected in the tree. The sections below list the metadata fields that are displayed on each tab.

Properties Tab

Displays properties of the selected item

Item	Applicable to types	Notes
Name	All	
GUID	Assembly	
Version	Assembly	
Culture	Assembly	
Revision	Assembly	
Build Number	Assembly	
Namespace	Class	
Assembly	Class	

ID	Class, Field, Property, Method, Event	The ID is an internal number that shows where to find the type in the assembly's internal metadata tables.
Extends	Class	The base class of the selected class
Extends ID	Class	The internal ID of the base class
Value Type	Field	
Value	Field	
Return Type	Method	

Attributes Tab

The **Attributes** tab shows all attributes (including custom attributes) that were applied to the selected item in source code. For each attribute, the name is displayed alongside the attribute's value.

Flags Tab

The **Flags** tab displays the set of metadata flags that could apply to the selected item. Each flag is represented by a check box. If the box is checked, the flag is set in the selected item's metadata. If the box is cleared, the flag has not been applied to the selected item.

Uses Tab

The **Uses** tab displays the list of assemblies that the selected assembly depends on. Each assembly listed must be deployed on the end-user's machine.

Implements Tab

The **Implements** tab is visible when the selected item is a class, sealed class, or interface. This tab lists each interface implemented by the selected item. Each implemented interface is a link that you can click. Clicking an implemented interface link will cause that item to be selected in the tree, and its metadata properties will be displayed. You can use the **Forward** and **Back** browser buttons on the **Toolbar** to quickly navigate back to the previously selected class or interface.

Implementors Tab

The **Implementors** tab is visible when an interface is selected in the left-hand pane. This tab displays all those classes that implement the interface.

Parameters Tab

The **Parameters** tab is visible when a method is selected in the left-hand pane. Each parameter is listed by name, alongside its type, and modifier (such as **ref** and **out**).

Call Graph Tab

The **Call Graph** tab is visible when a method is selected in the left-hand pane. This tab is divided into two panes: The top pane displays those methods that call the selected method. The bottom pane displays all the methods called by the selected method.

Certain methods, denoted by blue color and underlining, are clickable links; these are methods that are within the assembly you are viewing. Click a method link to make that method the selected item in the left-hand pane. Other methods listed in the **Calls** and **Called By** panes are not links; these methods are defined in assemblies outside the one you are viewing.

You can navigate forward and back to previously selected items using the **browser buttons** on the **Toolbar**.

See Also

Using COM Interop in Managed Applications

Exploring .NET Assembly Metadata (see page 158)

Adding a Reference to a COM Server

Adding an ActiveX Control to the Tool Palette

3.2.18.2 Type Library Explorer

File ▶ Open...

Use the type library explorer to inspect types and interfaces defined within a Windows type library.

Type Library Element	Icon	Page of Type Information	Contents of Page
Type Library	❖	**Attributes**	Name, version, GUID, and registered location of the type library. Also displays help context and help file information.
		Uses	A table showing the name and GUID of dependant type libraries.
		Flags	Flags that determine how other applications can use the type library.
CoClass	●	**Attributes**	Name, GUID, version, help context, and help file information.
		Flags	Flags that indicate how clients can create and use instances, whether the CoClass is visible in a browser, whether the CoClass is an ActiveX Control, and whether it can be aggregated.
		Implements	A table listing the interfaces (along with their attributes) the class implements.
Interface	🔑	**Attributes**	Name, version, GUID, help context, and help file information.
		Flags	Flags indicating whether the interface is hidden, dual, automation-compatible, and/or extensible.
DispInterface	🔑	**Attributes**	Name, version, GUID, help context, and help file information.
		Flags	Flags indicating whether the DispInterface is hidden, dual, automation-compatible, and/or extensible.
Method Method with return value	◆ ◆	**Attributes**	Name, dispatch ID, vtable offset, help string, and help context information.
		Flags	Flags indicating how clients can view and use the method, whether it is a default method for the interface, and whether it is replacable.
		Parameters	A table showing the name and type of all parameters on the method, and the return value if applicable.
SetByRef Method	◆	**Attributes**	ToDo
Getter Method Setter Method	◆ ◆	**Attributes**	Name, dispatch ID, vtable offset, help string, help context information.
		Flags	Flags indicating how the client can view and use the method, and whether it is hidden and/or browsable.
		Parameters	A table showing the name and type of all parameters on the method.

DispProperty		**Attributes**	Name, help string, and help context information.
		Flags	Flags to indicate how clients can view and use the property, whether it is a default property for the interface, and so on.
Alias		**Attributes**	Name, version, GUID, the type the alias represents, and help context information.
Record		**Attributes**	Name, version, GUID, help string and help context information.
Union		**Attributes**	Name, version, GUID, help string and help context information.
Enumeration		**Attributes**	Name, version, GUID, help string, and help context information.
Module		**Attributes**	Name, version, GUID, associated DLL name, help string, and help context information.
Field		**Attributes**	Name, type information, help string, and help context information.
		Flags	Flags indicating how clients can view and use the field, whether the field has a default value, and whether the field is bindable.
Constant		**Attributes**	Name, value, type (for module constants), help string, and help context information.
		Flags	Flags indicating how clients can view and use the constant, whether it represents a default value, and whether it is bindable.

See Also

Using COM Interop in Managed Applications

Adding a Reference to a COM Server

Adding an ActiveX Control to the Tool Palette

3.2.18.3 Search

Use this dialog box to search for various kinds of elements (such as classes or interfaces), or for a specific element within a .NET assembly, or a type library.

Item	Description
Text to find	The text string to search for in the assembly or type library.
Type to find	A drop-down list containing the types of elements you can search for. These are: Class, Interface, Method, Property, Field, and Event. If no element type is specified, all elements matching the search string will be returned.
Case sensitive	If checked, elements matching the search string must also match case. If unchecked, all elements matching the search string will be returned, regardless of case.
Search	Click to begin the search.

Items matching the search criteria are displayed in a table in the **Search** dialog. Items in the table are clickable links; clicking on an item will cause that item to be selected in the left-hand pane of the explorer view.

Tip: The Search

dialog is modeless, so you can leave it open and continue working with the main explorer window.

See Also

Using COM Interop in Managed Applications

Adding a Reference to a COM Server

Adding an ActiveX Control to the Tool Palette

3.3 Keyboard Mappings

The following topics list the keyboard mappings available in RAD Studio. Use the **Tools ▶ Options ▶ Editor Options ▶ Key Mappings** page to change the default keyboard mapping.

Topics

Name	Description
Key Mappings (↗ see page 1068)	**Tools ▶ Options ▶ Editor Options ▶ Key Mappings** Use this page to enable or disable key binding enhancement modules and change the order in which they are initialized.
BRIEF Keyboard Shortcuts (↗ see page 1069)	The following table lists the BRIEF Mapping keyboard shortcuts for the Code Editor.
IDE Classic Keyboard Shortcuts (↗ see page 1070)	The following table lists the IDE Classic Mapping keyboard shortcuts for the Code Editor. **Note:** Keyboard shortcuts that include the CTRL+ALT key combination are disabled when the **Use CTRL+ALT Keys** option is unchecked on the **Tools ▶ Options ▶ Editor Options ▶ Key Mappings** page.
Default Keyboard Shortcuts (↗ see page 1073)	The following table lists the Default Mapping keyboard shortcuts for the Code Editor. **Note:** Keyboard shortcuts that include the CTRL+ALT key combination are disabled when the **Use CTRL+ALT Keys** option is unchecked on the **Tools ▶ Options ▶ Editor Options ▶ Key Mappings** page.
Epsilon Keyboard Shortcuts (↗ see page 1076)	The following table lists the Epsilon Mapping keyboard shortcuts for the Code Editor. **Note:** Keyboard shortcuts that include the CTRL+ALT key combination are disabled when the **Use CTRL+ALT Keys** option is unchecked on the **Tools ▶ Options ▶ Editor Options ▶ Key Mappings** page.
Visual Basic Keyboard Shortcuts (↗ see page 1078)	The following table lists the Visual Basic Mapping keyboard shortcuts for the Code Editor. **Note:** Keyboard shortcuts that include the CTRL+ALT key combination are disabled when the **Use CTRL+ALT Keys** option is unchecked on the **Tools ▶ Options ▶ Editor Options ▶ Key Mappings** page.
Visual Studio Keyboard Shortcuts (↗ see page 1079)	The following table lists the Visual Studio Mapping keyboard shortcuts for the Code Editor. **Note:** Keyboard shortcuts that include the CTRL+ALT key combination are disabled when the **Use CTRL+ALT Keys** option is unchecked on the **Tools ▶ Options ▶ Editor Options ▶ Key Mappings** page.

3.3.1 Key Mappings

Tools ▶ Options ▶ Editor Options ▶ Key Mappings

Use this page to enable or disable key binding enhancement modules and change the order in which they are initialized.

Item	Description
Key mapping modules	Lists the available key bindings. To set the default key binding, use the **Editor Options** page **Editor SpeedSettings** option.

Enhancement modules	Enhancement modules are special packages that are installed and registered and use the keyboard binding features that can be developed using the Open Tools API. You can create enhancement modules that contain new keystrokes or apply new operations to existing keystrokes. Once installed, the enhancement modules are displayed in the **Enhancement modules** list box. Clicking the check box next to the enhancement module enables it and unchecking it disables it. Key mapping defined in an installed and enabled enhancement module overrides any existing key mapping defined for that key in the key mapping module which is currently in effect.
Move Up	Moves the selected enhancement module up one level in the list.
Move Down	Moves the selected enhancement module down one level in the list.
Use `CTRL+ALT` Keys	If checked, the `CTRL+ALT` key combination is used in shortcuts throughout the IDE. If unchecked, those shortcuts are disabled and `CTRL+ALT` can be used to perform other functions, such as entering accent characters.

See Also

Default Key Mapping (↗ see page 1073)

IDE Classic Key Mapping (↗ see page 1070)

BRIEF Emulation Key Mapping (↗ see page 1069)

Epsilon Emulation Key Mapping (↗ see page 1076)

Visual Studio Key Mapping (↗ see page 1079)

Visual Basic Key Mapping (↗ see page 1078)

3.3.2 BRIEF Keyboard Shortcuts

The following table lists the BRIEF Mapping keyboard shortcuts for the Code Editor.

Shortcut	Action
Alt+A	Marks a non-inclusive block
Alt+B	Displays a list of open files
Alt+Backspace	Deletes the word to the right of the cursor
Alt+C	Mark the beginning of a column block
Alt+D	Deletes a line
Alt+F9	Displays the local menu
Alt+Hyphen	Jumps to the previous page
Alt+I	Toggles insert mode
Alt+K	Deletes of the end of a line
Alt+L	Marks a line
Alt+M	Marks an inclusive block
Alt+N	Displays the contents of the next page
Alt+P	Prints the selected block
Alt+Page Down	Goes to the next tab
Alt+Page Up	Goes to the previous tab

Alt+Q	Causes next character to be interpreted as an ASCII sequence
Alt+R	Reads a block from a file
Backspace	Deletes the character to the left of the cursor
Ctrl+/	Adds or removes `//` to each line in the selected code block to comment the code.
Ctrl+- (dash)	Closes the current page
Ctrl+B	Moves to the bottom of the window
Ctrl+Backspace	Deletes the word to the left of the cursor
Ctrl+C	Centers line in window
Ctrl+D	Moves down one screen
Ctrl+E	Moves up one screen
Ctrl+Enter	Inserts an empty new line
Ctrl+F1	Help keyword search
Ctrl+F5	Toggles case-sensitive searching
Ctrl+F6	Toggles regular expression searching
Ctrl+K	Deletes to the beginning of a line
Ctrl+M	Inserts a new line with a carriage return
Ctrl+O+A	Open file at cursor
Ctrl+O+B	Browse symbol at cursor
Ctrl+O+O	Toggles the case of a selection
Ctrl+Q+[	Finds the matching delimiter (forward)
Ctrl+Q+]	Finds the matching delimiter (backward)
Ctrl+Q+Ctrl+[	Finds the matching delimiter (forward)
Ctrl+Q+Ctrl+]	Finds the matching delimiter (backward)
Ctrl+S	Performs an incremental search
Ctrl+T	Moves to the top of the window
Ctrl+Shift+C	Invokes class completion for the class declaration in which the cursor is positioned
Del	Deletes a character or block at the cursor
Enter	Inserts a new line with a carriage return
Esc	Cancels a command at the prompt
Shift+Backspace	Deletes the character to the left of the cursor
Shift+F4	Tiles windows horizontally
Shift+F6	Repeats the last Search|Replace operation
Tab	Inserts a tab character

3.3.3 IDE Classic Keyboard Shortcuts

The following table lists the IDE Classic Mapping keyboard shortcuts for the Code Editor.

Note: Keyboard shortcuts that include the CTRL+ALT

key combination are disabled when the **Use CTRL+ALT Keys** option is unchecked on the **Tools**▶**Options**▶**Editor Options**▶**Key Mappings** page.

Shortcut	Action
Alt+[	Finds the matching delimiter (forward)
Alt+]	Finds the matching delimiter (backward)
Alt+Page Down	Goes to the next tab
Alt+Page Up	Goes to the previous tab
Alt+Shift+Down Arrow	Moves the cursor down one line and selects the column from the left of the starting cursor position
Alt+Shift+End	Selects the column from the cursor position to the end of the current line
Alt+Shift+Home	Selects the column from the cursor position to the start of the current line
Alt+Shift+Left Arrow	Selects the column to the left of the cursor
Alt+Shift+Page Down	Moves the cursor down one line and selects the column from the right of the starting cursor position
Alt+Shift+Page Up	Moves the cursor up one screen and selects the column from the left of the starting cursor position
Alt+Shift+Right Arrow	Selects the column to the right of the cursor
Alt+Shift+Up Arrow	Moves the cursor up one line and selects the column from the left of the starting cursor position
Click+Alt+mousemove	Selects column-oriented blocks
Ctrl+/	Adds or removes `//` to each line in the selected code block to comment the code.
Ctrl+Alt+Shift+End	Selects the column from the cursor position to the end of the current file
Ctrl+Alt+Shift+Home	Selects the column from the cursor position to the start of the current file
Ctrl+Alt+Shift+Left Arrow	Selects the column to the left of the cursor
Ctrl+Alt+Shift+Page Down	Selects the column from the cursor position to the top of the screen
Ctrl+Alt+Shift+Page Up	Selects the column from the cursor position to the bottom of the screen
Ctrl+Alt+Shift+Right Arrow	Selects the column to the right of the cursor
Ctrl+Backspace	Deletes the word to the right of the cursor
Ctrl+Del	Deletes a currently selected block
Ctrl+Down Arrow	Scrolls down one line
Ctrl+End	Moves to the end of a file
Ctrl+Enter	Opens file at cursor
Ctrl+Home	Moves to the top of a file
Ctrl+I	Inserts a tab character
Ctrl+J	Templates pop-up menu
Ctrl+Left Arrow	Moves one word left
Ctrl+N	Inserts a new line
Ctrl+O+C	Turns on column blocking
Ctrl+O+K	Turns off column blocking
Ctrl+O+O	Insert compiler options

Ctrl+P	Causes next character to be interpreted as an ASCII sequence
Ctrl+PgDn	Moves to the bottom of a screen
Ctrl+PgUp	Moves to the top of a screen
Ctrl+Right Arrow	Moves one word right
Ctrl+Shift+C	Invokes class completion for the class declaration in which the cursor is positioned
Ctrl+Shift K+A	Expands all blocks of code
Ctrl+Shift K+E	Collapses a block of code
Ctrl+Shift K+O	Toggles between enabling and disabling Code Folding
Ctrl+Shift K+T	Toggles the current block between collapsed and expanded
Ctrl+Shift K+U	Expands a block of code
Ctrl+Shift+End	Selects from the cursor position to the end of the current file
Ctrl+Shift+G	Inserts a new Globally Unique Identifier (GUID)
Ctrl+Shift+Home	Selects from the cursor position to the start of the current file
Ctrl+Shift+I	Indents block
Ctrl+Shift+Left Arrow	Selects the word to the left of the cursor
Ctrl+Shift+PgDn	Selects from the cursor position to the bottom of the screen
Ctrl+Shift+PgUp	Selects from the cursor position to the top of the screen
Ctrl+Shift+Right Arrow	Selects the word to the right of the cursor
Ctrl+Shift+space bar	Code Parameters pop-up window
Ctrl+Shift+Tab	Moves to the previous code page (or file)
Ctrl+Shift+Tab	Moves to the previous page
Ctrl+Shift+U	Outdents block
Ctrl+Shift+Y	Deletes to the end of a line
Ctrl+space bar	Code Completion pop-up window
Ctrl+T	Deletes a word
Ctrl+Tab	Moves to the next code page (or file)
Ctrl+Up Arrow	Scrolls up one line
Ctrl+Y	Deletes a line
F1	Displays Help for the selected fully qualified namespace
Shift+Alt+arrow	Selects column-oriented blocks
Shift+Backspace	Deletes the character to the left of the cursor
Shift+Down Arrow	Moves the cursor down one line and selects from the right of the starting cursor position
Shift+End	Selects from the cursor position to the end of the current line
Shift+Enter	Inserts a new line with a carriage return
Shift+Home	Selects from the cursor position to the start of the current line
Shift+Left Arrow	Selects the character to the left of the cursor
Shift+PgDn	Moves the cursor down one line and selects from the right of the starting cursor position
Shift+PgUp	Moves the cursor up one screen and selects from the left of the starting cursor position
Shift+Right Arrow	Selects the character to the right of the cursor

Shift+Space	Inserts a blank space
Shift+Tab	Moves the cursor to the left one tab position
Shift+Up Arrow	Moves the cursor up one line and selects from the left of the starting cursor position

3.3.4 Default Keyboard Shortcuts

The following table lists the Default Mapping keyboard shortcuts for the Code Editor.

Note: Keyboard shortcuts that include the CTRL+ALT

key combination are disabled when the **Use CTRL+ALT Keys** option is unchecked on the **Tools**▶**Options**▶**Editor Options**▶**Key Mappings** page.

Shortcut	**Action**
Alt+[	Finds the matching delimiter (forward).
Alt+]	Finds the matching delimiter (backward).
Alt+Left Arrow	Go back after **Alt+Up Arrow** or **Ctrl+Click** (go to declaration) operation.
Alt+F7	Go to previous error or message in **Message View**.
Alt+F8	Go to next error / message in **Message View**.
Alt+Page Down	Goes to the next tab.
Alt+Page Up	Goes to the previous tab.
Alt+Right Arrow	Go forward after **Alt+Left Arrow** operation.
Alt+Shift+Down Arrow	Moves the cursor down one line and selects the column from the left of the starting cursor position.
Alt+Shift+End	Selects the column from the cursor position to the end of the current line.
Alt+Shift+Home	Selects the column from the cursor position to the start of the current line.
Alt+Shift+Left Arrow	Selects the column to the left of the cursor.
Alt+Shift+Page Down	Moves the cursor down one line and selects the column from the right of the starting cursor position.
Alt+Shift+Page Up	Moves the cursor up one screen and selects the column from the left of the starting cursor position.
Alt+Shift+Right Arrow	Selects the column to the right of the cursor.
Alt+Shift+Up Arrow	Moves the cursor up one line and selects the column from the left of the starting cursor position.
Alt+Up Arrow	Go to declaration.
Click+Alt+mousemove	Selects column-oriented blocks.
Ctrl+/	Adds or removes `//` to each line in the selected code block to comment the code.
Ctrl+Alt+F12	Display a drop down list of open files.
Ctrl+Alt+Shift+End	Selects the column from the cursor position to the end of the current file.
Ctrl+Alt+Shift+Home	Selects the column from the cursor position to the start of the current file.
Ctrl+Alt+Shift+Left Arrow	Selects the column to the left of the cursor.
Ctrl+Alt+Shift+Page Down	Selects the column from the cursor position to the top of the screen.

Ctrl+Alt+Shift+Page Up	Selects the column from the cursor position to the bottom of the screen.
Ctrl+Alt+Shift+Right Arrow	Selects the column to the right of the cursor.
Ctrl+Backspace	Deletes the word to the right of the cursor.
Ctrl+Click	Go to declaration.
Ctrl+Del	Deletes a currently selected block.
Ctrl+Down Arrow	Scrolls down one line.
Ctrl+End	Moves to the end of a file.
Ctrl+Enter	Opens file at cursor.
Ctrl+Home	Moves to the top of a file.
Ctrl+I	Inserts a tab character.
Ctrl+J	Templates pop-up menu.
Ctrl+K+*n*	Sets a bookmark, where *n* is a number from 0 to 9.
Ctrl+K+T	Select word.
Ctrl+Left Arrow	Moves one word left.
Ctrl+*n*	Jumps to a bookmark, where *n* is the number of the bookmark, from 0 to 9.
Ctrl+N	Inserts a new line.
Ctrl+O+C	Turns on column blocking.
Ctrl+O+K	Turns off column blocking.
Ctrl+O+L	Turn on line blocking mode.
Ctrl+O+O	Insert compiler options.
Ctrl+P	Causes next character to be interpreted as an ASCII sequence.
Ctrl+PgDn	Moves to the bottom of a screen.
Ctrl+PgUp	Moves to the top of a screen.
Ctrl+Q+#	Go to bookmark.
Ctrl+Right Arrow	Moves one word right.
Ctrl+Shift+C	Invokes class completion for the class declaration in which the cursor is positioned.
Ctrl+Shift+#	Set bookmark.
Ctrl+Shift+B	Display buffer list.
Ctrl+Shift+Down Arrow	Jump between declaration and implementation.
Ctrl+Shift+Enter	Find usages.
Ctrl+Shift+J	SyncEdit.
Ctrl+Shift K+A	Expands all blocks of code.
Ctrl+Shift K+C	Collapses all classes.
Ctrl+Shift K+E	Collapses a block of code.
Ctrl+Shift K+G	Initializes/finalize or interface/implementation.
Ctrl+Shift K+M	Collapses all methods.
Ctrl+Shift K+N	Collapses namespace/Unit.
Ctrl+Shift K+O	Toggles between enabling and disabling Code Folding.

Ctrl+Shift K+P	Collapses nested procedures.
Ctrl+Shift K+R	Collapses all regions.
Ctrl+Shift K+T	Toggles the current block between collapsed and expanded.
Ctrl+Shift K+U	Expands a block of code.
Ctrl+Shift+End	Selects from the cursor position to the end of the current file.
Ctrl+Shift+G	Inserts a new Globally Unique Identifier (GUID).
Ctrl+Shift+Home	Selects from the cursor position to the start of the current file.
Ctrl+Shift+I	Indents block.
Ctrl+Shift+Left Arrow	Selects the word to the left of the cursor.
Ctrl+Shift+P	Plays a recorded keystroke macro.
Ctrl+Shift+PgDn	Selects from the cursor position to the bottom of the screen.
Ctrl+Shift+PgUp	Selects from the cursor position to the top of the screen.
Ctrl+Shift+R	Toggles between starting and stopping the recording of a keystroke macro.
Ctrl+Shift+Right Arrow	Selects the word to the right of the cursor.
Ctrl+Shift+space bar	Code Parameters pop-up window.
Ctrl+Shift+T	Create ToDo entry.
Ctrl+Shift+Tab	Moves to the previous code page (or file).
Ctrl+Shift+Tab	Moves to the previous page.
Ctrl+Shift+U	Outdents block.
Ctrl+Shift+Up Arrow	Jump between declaration and implementation.
Ctrl+Shift+Y	Deletes to the end of a line.
Ctrl+space bar	Code Completion pop-up window.
Ctrl+T	Deletes a word.
Ctrl+Tab	Moves to the next code page (or file).
Ctrl+Up Arrow	Scrolls up one line.
Ctrl+Y	Deletes a line.
F1	Displays Help for the selected fully qualified namespace.
Shift+Alt+arrow	Selects column-oriented blocks.
Shift+Backspace	Deletes the character to the left of the cursor.
Shift+Down Arrow	Moves the cursor down one line and selects from the right of the starting cursor position.
Shift+End	Selects from the cursor position to the end of the current line.
Shift+Enter	Inserts a new line with a carriage return.
Shift+Home	Selects from the cursor position to the start of the current line.
Shift+Left Arrow	Selects the character to the left of the cursor.
Shift+PgDn	Moves the cursor down one line and selects from the right of the starting cursor position.
Shift+PgUp	Moves the cursor up one screen and selects from the left of the starting cursor position.
Shift+Right Arrow	Selects the character to the right of the cursor.
Shift+Space	Inserts a blank space.
Shift+Tab	Moves the cursor to the left one tab position.

Shift+Up Arrow	Moves the cursor up one line and selects from the left of the starting cursor position.

3.3.5 Epsilon Keyboard Shortcuts

The following table lists the Epsilon Mapping keyboard shortcuts for the Code Editor.

Note: Keyboard shortcuts that include the CTRL+ALT

key combination are disabled when the **Use CTRL+ALT Keys** option is unchecked on the **Tools**▶**Options**▶**Editor Options**▶**Key Mappings** page.

Shortcut	**Action**
Alt+)	Locates the next matching delimiter (cursor must be on ')', '}' or ']')
Alt+?	Displays context-sensitive Help
Alt+\	Deletes spaces and tabs around the cursor on the same line
Alt+Backspace	Deletes the word to the left of the current cursor position
Alt+C	Capitalizes the first letter of the character after the cursor and lowercases all other letters to the end of the word
Alt+D	Deletes to word to the right of the cursor
Alt+Del	Deletes all text in the block between the cursor and the previous matching delimiter (cursor must be on ')', '}' or ']')
Alt+L	Converts the current word to lowercase
Alt+Shift+/	Displays context-sensitive Help
Alt+Shift+O	Locates the next matching delimiter (cursor must be on ')', '}' or ']')
Alt+T	Transposes the two words on either side of the cursor
Alt+Tab	Indents to the current line to the text on the previous line
Alt+U	Converts a selected word to uppercase or converts from the cursor position to the end of the word to uppercase
Alt+X	Invokes the specified command or macro
Backspace	Deletes the character to the left of the current cursor position
Ctrl+/	Adds or removes `//` to each line in the selected code block to comment the code.
Ctrl+_	Displays context-sensitive Help
Ctrl+Alt+B	Locates the next matching delimiter (cursor must be on ')', '}' or ']')
Ctrl+Alt+F	Locates the previous matching delimiter (cursor must be on ')', '}' or ']')
Ctrl+Alt+H	Deletes the word to the left of the current cursor position
Ctrl+Alt+K	Deletes all text in the block between the cursor and the next matching delimiter (cursor must be on ')', '}' or ']')
Ctrl+D	Deletes the currently selected character or character to the right of the cursor
Ctrl+H	Deletes the character to the left of the current cursor position
Ctrl+K	Cuts the contents of line and places it in the clipboard
Ctrl+L	Centers the active window

Ctrl+M	Inserts a carriage return
Ctrl+O	Inserts a new line after the cursor
Ctrl+Q	Interpret next character as an ASCII code
Ctrl+R	Incrementally searches backward through the current file
Ctrl+S	Incrementally searches for a string entered from the keyboard
Ctrl+Shift+-	Displays context-sensitive Help
Ctrl+Shift+C	Invokes class completion for the class declaration in which the cursor is positioned
Ctrl+T	Transposes the two characters on either side of the cursor
Ctrl+X+,	Browses the symbol at the cursor
Ctrl+X+0	Deletes the contents of the current window
Ctrl+X+Ctrl+E	Invoke a command processor
Ctrl+X+Ctrl+T	Transposes the two lines on either side of the cursor
Ctrl+X+Ctrl+X	Exchanges the locations of the cursor position and a bookmark
Ctrl+X+I	Inserts the contents of a file at the cursor
Ctrl+X+N	Displays the next window in the buffer list
Ctrl+X+P	Displays the previous window in the buffer list
Esc+)	Locates the next matching delimiter (cursor must be on ')', '}' or ']')
Esc+?	Displays context-sensitive Help
Esc+\	Deletes spaces and tabs around the cursor on the same line
Esc+BackSpace	Deletes the word to the left of the current cursor position
Esc+C	Capitalizes the first letter of the character after the cursor and lowercases all other letters to the end of the word
Esc+Ctrl+B	Locates the next matching delimiter (cursor must be on ')', '}' or ']')
Esc+Ctrl+F	Locates the previous matching delimiter (cursor must be on ')', '}' or ']')
Esc+Ctrl+H	Deletes the word to the left of the current cursor position
Esc+Ctrl+K	Deletes all text in the block between the cursor and the next matching delimiter (cursor must be on ')', '}' or ']')
Esc+D	Deletes to word to the right of the cursor
Esc+Del	Deletes all text in the block between the cursor and the previous matching delimiter (cursor must be on ')', '}' or ']')
Esc+End	Displays the next window in the buffer list
Esc+Home	Displays the previous window in the buffer list
Esc+L	Converts the current word to lowercase
Esc+T	Transposes the two words on either side of the cursor
Esc+Tab	Indents to the current line to the text on the previous line
Esc+U	Converts a selected word to uppercase or converts from the cursor position to the end of the word to uppercase
Esc+X	Invokes the specified command or macro
F2	Invokes the specified command or macro

3.3.6 Visual Basic Keyboard Shortcuts

The following table lists the Visual Basic Mapping keyboard shortcuts for the Code Editor.

Note: Keyboard shortcuts that include the CTRL+ALT

key combination are disabled when the **Use CTRL+ALT Keys** option is unchecked on the **Tools**▶**Options**▶**Editor Options**▶**Key Mappings** page.

Alt+[	Finds the matching delimiter (forward)
Alt+]	Finds the matching delimiter (backward)
Alt+F12	Browse symbol at cursor (Delphi)
Alt+Page Down	Goes to the next tab
Alt+Page Up	Goes to the previous tab
Alt+Shift+Backspace	Edit|Redo
Alt+Shift+Down Arrow	Moves the cursor down one line and selects the column from the left of the starting cursor position
Alt+Shift+End	Selects the column from the cursor position to the end of the current line
Alt+Shift+Home	Selects the column from the cursor position to the start of the current line
Alt+Shift+Left Arrow	Selects the column to the left of the cursor
Alt+Shift+Page Down	Moves the cursor down one line and selects the column from the right of the starting cursor position
Alt+Shift+Page Up	Moves the cursor up one screen and selects the column from the left of the starting cursor position
Alt+Shift+Right Arrow	Selects the column to the right of the cursor
Alt+Shift+Up Arrow	Moves the cursor up one line and selects the column from the left of the starting cursor position
Backspace	Deletes the character to the left of the cursor
Ctrl+Alt+Shift+End	Selects the column from the cursor position to the end of the current file
Ctrl+Alt+Shift+Home	Selects the column from the cursor position to the start of the current file
Ctrl+Alt+Shift+Left Arrow	Selects the column to the left of the cursor
Ctrl+Alt+Shift+Page Down	Selects the column from the cursor position to the top of the screen
Ctrl+Alt+Shift+Page Up	Selects the column from the cursor position to the bottom of the screen
Ctrl+Alt+Shift+Right Arrow	Selects the column to the right of the cursor
Ctrl+Backspace	Deletes the word to the left of the cursor
Ctrl+F4	Closes the current page
Ctrl+G	Open file at cursor
Ctrl+j	Templates pop-up menu
Ctrl+K+C	Adds or removes `//` to each line in the selected code block to comment the code.
Ctrl+K+E	Converts the word under the cursor to lower case
Ctrl+K+F	Converts the word under the cursor to upper case
Ctrl+L	Deletes a line

Ctrl+P	Causes next character to be interpreted as an ASCII sequence
Ctrl+Q+[	Finds the matching delimiter (forward)
Ctrl+Q+]	Finds the matching delimiter (backward)
Ctrl+Q+Ctrl+[	Finds the matching delimiter (forward)
Ctrl+Q+Ctrl+]	Finds the matching delimiter (backward)
Ctrl+Q+Y	Deletes to the end of a line
Ctrl+Shift+C	Invokes class completion for the class declaration in which the cursor is positioned
Ctrl+Shift+End	Selects from the cursor position to the end of the current file
Ctrl+Shift+Home	Selects from the cursor position to the start of the current file
Ctrl+Shift+Left Arrow	Selects the word to the left of the cursor
Ctrl+Shift+PgDn	Selects from the cursor position to the bottom of the screen
Ctrl+Shift+PgUp	Selects from the cursor position to the top of the screen
Ctrl+Shift+Right Arrow	Selects the word to the right of the cursor
Ctrl+Shift+Tab	Displays the previous window in the buffer list
Ctrl+T	Deletes the word to the left of the cursor
Ctrl+Tab	Displays the next window in the buffer list
Ctrl+y	Deletes a line
Ctrl+Y	Deletes to the end of a line
Delete	Deletes a character or block at the cursor
Enter	Inserts a new line character
Insert	Toggles insert mode
Shift+Backspace	Deletes the character to the left of the cursor
Shift+Down Arrow	Moves the cursor down one line and selects from the right of the starting cursor position
Shift+End	Selects from the cursor position to the end of the current line
Shift+Enter	Inserts a new line character
Shift+Home	Selects from the cursor position to the start of the current line
Shift+Left Arrow	Selects the character to the left of the cursor
Shift+PgDn	Moves the cursor down one line and selects from the right of the starting cursor position
Shift+PgUp	Moves the cursor up one screen and selects from the left of the starting cursor position
Shift+Right Arrow	Selects the character to the right of the cursor
Shift+Space	Inserts a blank space
Shift+Up Arrow	Moves the cursor up one line and selects from the left of the starting cursor position
Tab	Inserts a tab character

3.3.7 Visual Studio Keyboard Shortcuts

The following table lists the Visual Studio Mapping keyboard shortcuts for the Code Editor.

Note: Keyboard shortcuts that include the CTRL+ALT

key combination are disabled when the **Use CTRL+ALT Keys** option is unchecked on the **Tools**▶**Options**▶**Editor Options**▶**Key Mappings** page.

Shortcut	Action
Alt+[	Finds the matching delimiter (forward)
Alt+]	Finds the matching delimiter (backward)
Alt+Backspace	Edit|Undo
Alt+F12	Browse symbol at cursor (Delphi)
Alt+Page Down	Goes to the next tab
Alt+Page Up	Goes to the previous tab
Alt+Shift+Backspace	Edit|Redo
Alt+Shift+Down Arrow	Moves the cursor down one line and selects the column from the left of the starting cursor position
Alt+Shift+End	Selects the column from the cursor position to the end of the current line
Alt+Shift+Home	Selects the column from the cursor position to the start of the current line
Alt+Shift+Left Arrow	Selects the column to the left of the cursor
Alt+Shift+Page Down	Moves the cursor down one line and selects the column from the right of the starting cursor position
Alt+Shift+Page Up	Moves the cursor up one screen and selects the column from the left of the starting cursor position
Alt+Shift+Right Arrow	Selects the column to the right of the cursor
Alt+Shift+Up Arrow	Moves the cursor up one line and selects the column from the left of the starting cursor position
Ctrl+Alt+Shift+End	Selects the column from the cursor position to the end of the current file
Ctrl+Alt+Shift+Home	Selects the column from the cursor position to the start of the current file
Ctrl+Alt+Shift+Left Arrow	Selects the column to the left of the cursor
Ctrl+Alt+Shift+Page Down	Selects the column from the cursor position to the top of the screen
Ctrl+Alt+Shift+Page Up	Selects the column from the cursor position to the bottom of the screen
Ctrl+Alt+Shift+Right Arrow	Selects the column to the right of the cursor
Ctrl+F4	Closes the current page
Ctrl+J	Templates pop-up menu
Ctrl+K+C	Adds or removes `//` to each line in the selected code block to comment the code.
Ctrl+K+E	Converts the word under the cursor to lower case
Ctrl+K+F	Converts the word under the cursor to upper case
Ctrl+L	Search|Search Again
Ctrl+P	Causes next character to be interpreted as an ASCII sequence
Ctrl+Q+[	Finds the matching delimiter (forward)
Ctrl+Q+]	Finds the matching delimiter (backward)
Ctrl+Q+Ctrl+[	Finds the matching delimiter (forward)
Ctrl+Q+Ctrl+]	Finds the matching delimiter (backward)
Ctrl+Q+Y	Deletes to the end of a line

Ctrl+Shift+C	Invokes class completion for the class declaration in which the cursor is positioned
Ctrl+Shift+End	Selects from the cursor position to the end of the current file
Ctrl+Shift+Home	Selects from the cursor position to the start of the current file
Ctrl+Shift+Left Arrow	Selects the word to the left of the cursor
Ctrl+Shift+PgDn	Selects from the cursor position to the bottom of the screen
Ctrl+Shift+PgUp	Selects from the cursor position to the top of the screen
Ctrl+Shift+Right Arrow	Selects the word to the right of the cursor
Ctrl+Shift+Tab	Displays the previous window in the buffer list
Ctrl+T	Deletes the word to the left of the cursor
Ctrl+Tab	Displays the next window in the buffer list
Ctrl+Y	Deletes to the end of a line
Shift+Down Arrow	Moves the cursor down one line and selects from the right of the starting cursor position
Shift+End	Selects from the cursor position to the end of the current line
Shift+Enter	Inserts a new line character
Shift+Home	Selects from the cursor position to the start of the current line
Shift+Left Arrow	Selects the character to the left of the cursor
Shift+PgDn	Moves the cursor down one line and selects from the right of the starting cursor position
Shift+PgUp	Moves the cursor up one screen and selects from the left of the starting cursor position
Shift+Right Arrow	Selects the character to the right of the cursor
Shift+Space	Inserts a blank space
Shift+Up Arrow	Moves the cursor up one line and selects from the left of the starting cursor position

3.4 Command Line Switches and File Extensions

The following topic lists the IDE command line switches and options.

Topics

Name	Description
IDE Command Line Switches and Options (↗ see page 1082)	Describes available options when starting the IDE from the command line.
File Extensions of Files Generated by RAD Studio (↗ see page 1084)	The following table lists the extensions of files generated by RAD Studio. **Note:** MSBuild requires that the extension for all project files end in 'proj' (that is, MSBuild uses the mask *.*proj).

3.4.1 IDE Command Line Switches and Options

Describes available options when starting the IDE from the command line.

IDE command line switches

The following options are available when starting the IDE from the command line. You must precede all options (unless otherwise noted) with either a dash (-) or a slash (/). The options are not case-sensitive. Therefore, the following options are all identical:

`-d /d -D /D`.

Use the IDE command line switches with the IDE startup command: `bds.exe`

For example:

Code	Does this
bds.exe -ns	Starts the RAD Studio IDE with no splash screen.
bds.exe &#x2011;sdc:\test\source -d c:\test\myprog.exe \mbox{-}td	Starts the RAD Studio IDE and loads c:\test\myprog.exe into the debugger and uses c:\test\source as the location for the source code while debugging. The `-td` and any other argument that appears after the debugger option (`-d`*exename*) is used as an argument to c:\test\myprog.exe.

General options

Option	Description
?	Launches the IDE and displays online help for IDE command-line options.
-- (two hyphens)	Ignore rest of command-line.
ns	No splash screen. Suppresses display of the splash screen during IDE startup.
np	No welcome page. Does not display the welcome page after starting the IDE.

p*personality*	Starts the specified personality of the RAD Studio IDE. The possible values for personality are:
	`Delphi`
	`CBuilder`
	`DelphiDotNet`

rregkey	Allows you to specify an alternate base registry key so you can run two copies of the IDE using different configurations. This allows component developers to debug a component at design-time by using the IDE as the hosting application without the debugging IDE interfering by trying to load the component package being developed.

Debugger options

Option	Description
attach:%1;%2	Performs a debug attach, using %1 as the process ID to attach to and %2 as the event ID for that process. The `attach` option can be used manually, but is used mostly for Just in Time debugging.
d*exename*	Loads the specified executable (exename) into the debugger. Any parameters specified after the exename are used as parameters to the program being debugged and are ignored by the IDE. A space is allowed between the `-d` and the exename.

The following options can only be used with the `-d` option:

debugger=[borwin32|bordotnet]	Indicates which debugger to use. • `Borwin32` invokes the CodeGear Win32 Debugger. • `Bordotnet` invokes the CodeGear .NET Debugger. If this option is omitted, the debugger that was first registered in the IDE is used. You can use this particular switch with the `attach` option as well as the `-d` option.
l	(Lowercase L) Assembler startup. Do not execute startup code. Must be used with the d option. Normally, when you specify the `-d` option, the debugger attempts to run the process to either main or WinMain. When `-l` is specified, the process is merely loaded and no startup code is executed.
sd*directories*	Source Directories. Must be used with the `-d` option. The argument is either a single directory or a semicolon delimited list of directories which are used as the **Debug Source Path** setting (can also be set using the **Project**▶**Options**▶**Debugger** page). No space is allowed between sd and the directory list argument.
h*hostname*	Remote debugger host name. Must be used with the `-d` option. A remote debug session is initiated using the specified host name as the remote host where debugging is performed. The remote debug server program must be running on the remote host.
tworkingdirectory	Specifies a working directory for your debug session (corresponds to "Working directory" setting on the Load Process dialog).

Project options

Option	Description
filename	(No preceding dash) The specified filename is loaded in the IDE. It can be a project, project group, or a single file.
b	AutoBuild. Must be used with the `filename` option. When the `-b` option is specified, the project or project group is built automatically when the IDE starts. Any hints, errors, or warnings are then saved to a file. Then the IDE exits. This facilitates doing builds in batch mode from a batch file. The Error Level is set to 0 for successful builds and 1 for failed builds. By default, the output file has the same name as the filename specified with the file extension changed to `.err`. This can be overridden using the `-o` option.
m	AutoMake. Same as AutoBuild, but a make is performed rather than a full build.
ooutputfile	Output file. Must be used the `-b` or `-m` option. When the `-o` option is specified, any hints, warnings, or errors are written to the file specified instead of the default file.

3.4.2 File Extensions of Files Generated by RAD Studio

The following table lists the extensions of files generated by RAD Studio.

Note: MSBuild requires that the extension for all project files end in 'proj' (that is, MSBuild uses the mask *.*proj).

File Extension	Description
app.config	Dynamic properties and their values for .NET Windows applications.
asax, ascx, asmx, aspx	ASP.NET application files.
bdsproj	Project options file for BDS 2006 and earlier. Contains the current settings for project options, such as compiler and linker settings, directories, conditional directives, and command-line parameters. Set these options using **Project ▶ Options**.
bdsgroup	Project group for BDS 2006 and earlier products.
bpk	Source file for a C++ Builder package; produces a .bpl file when compiled and linked.
bpl	A compiled Delphi package or a compiled C++ package (see also `.dpk`).
bpr	C++ Builder project source; when compiled, produces .exe, .dll or .ocx file.
cbproj	C++ project. This is not 'cproj', which is a VS.NET C++ project.
cfg	Project configuration file used for command line compiles. The compiler searches for a `dcc32.cfg` in the compiler executable directory, then for `dcc32.cfg` in the current directory, and then finally for `projectname.cfg` in the project directory. You can type `dcc32 projectname` on the command line and compile the project with same options specified in the IDE.
config	Config files contain project option information and build logic. Each project has a .config file.
cpp	C++ source file.
cs	The code-behind file for ASP.NET; also a C# source code file.
csm	C++ precompiled header file.
csproj	C# project, in keeping with MSBuild convention.
dci	Holds Code Insight changes you make in the IDE.
dcp	Contains all compile and link information for a Delphi package in the same way that a .dcu file has this information for a .pas file. Use this file if building with runtime packages.
dcpil	Delphi.NET compiled CLR import. Same as .dcp file for .NET.
dcu	Delphi compiled unit. An intermediate compiler output file produced for each Win32 unit's source code. You do not need to open these binary files or distribute them with your application. The `.dcu` file enables rapid compiling and linking.
dcuil	An intermediate compiler output file produced for each .NET unit's source code. Same as .dcu file for .NET.
delphi.dct	Component template changes you make in the IDE.
dfm	A Windows VCL form file.
dpk	The source file for a Delphi package. When compiled, produces a `.bpl` file.
dpr	Delphi project source; when compiled produces .exe, .dll, or .ocx file.

dproj	Delphi project for both native and .NET.
dsk	File used to save the desktop when the Autosave Project desktop option is On.
dst	File used to save the desktop speed setting as set in the IDE toolbar desktop combo box.
exe	Executable file.
exe.incr	Incremental build information.
groupproj	Project group.
h	C++ header source file.
hpp	Pascal generated C++ header file.
i	C++ preprocessor output (not saved by default). Each .cpp and all of its included headers are preprocessed into a .i file,
identcache	Information used for refactoring.
il?	C++ incremental linking state file.
nfm	A .NET VCL form file.
nfn	A file maintained by the Translation Tools, containing translated strings and other data displayed in the Translation Manager. There is a separate `.nfn` file for each form in your application and each target language.
obj	C++ compiled translation unit. Each .cpp and all of its included header are compiled into a resultant .obj file.
optset	Named option set file that stores configuration options, separately from projects.
pas	Delphi (Pascal) source code file.
pdb	Contains project debugging information for .NET.
res, rc	Compiled and uncompiled resource files.
resources	A binary resource file that can be embedded in a runtime executable or compiled into satellite assembly for .NET.
resx	An XML resource file containing nodes that represent objects and strings for .NET.
rsp	Response file used by the C++ linker.
targets	Targets file, an MSBuild-compliant XML file you add to your project to allow customizing the build process. It contains MSBuild scripts among other information.
tlb	Type library.
todo	The project to-do list.
tvsconfig	Modeling configuration file.
txvpck, txvcls	Information for model diagram.
vproj	Visual Basic projects, in keeping with VS.NET and MSBuild convention.
web.config	Dynamic properties and their values for ASP.NET applications.
#nn	#nn = #00, #01, #02, and so forth. C++ precompiled header file.

See Also

Packages and Standard DLLs

3.5 Together Reference

This section contains links to the reference material for UML modeling with Together.

Topics

Name	Description
Together Configuration Options (↗ see page 1086)	This section describes UML modeling options.
Together Keyboard Shortcuts (↗ see page 1104)	Together enables you to perform many diagram actions without using the mouse. You can navigate between diagrams, create diagram elements, and more, using the keyboard only.
GUI Components for Modeling (↗ see page 1105)	This section describes GUI components of the RAD Studio interface you use for UML modeling.
Together Refactoring Operations (↗ see page 1115)	The following refactoring operations are available in Together: ***Refactoring operations***
Project Types and Formats with Support for Modeling (↗ see page 1116)	There are two basic project types: • **Design project.** Project file extension: `.bdsproj.tgproj`. These projects are language-neutral and comply with one of the two versions of UML specifications: UML 1.5 or UML 2.0. • **Implementation project.** Project file extension: `.bdsproj.csproj` (Visual C# .NET), and `.vbproj` (Visual Basic .NET). You can create models for language-specific projects. Modeling that complies with UML 1.5 specification is supported for C# and DelphiVisual Basic .NET projects. Together modeling features are automatically activated for these projects. The set of available project types depends on your license. Together Designer is required to work with... more (↗ see page 1116)
UML 1.5 Reference (↗ see page 1116)	This section contains reference material about UML 1.5 diagrams.
Together Glossary (↗ see page 1139)	This topic contains a dictionary of specific terms used in Together user interface and documentation. This dictionary is sorted alphabetically.
UML 2.0 Reference (↗ see page 1140)	This section contains reference material about UML 2.0 diagrams.
Together Wizards (↗ see page 1158)	This section describes wizards used for UML modeling.

3.5.1 Together Configuration Options

This section describes UML modeling options.

Topics

Name	Description
Configuration Levels (↗ see page 1088)	The configuration options apply to the four hierarchical levels. Each level contains a number of categories, or groups, of options. The configuration options can be specified at the following levels: • **Default**: Options at this level apply to all the current projects and project groupsolutions, as well as newly created ones. Diagram options apply to newly created diagrams within these projects and project groupsolutions. All options are available at this level. • **Project groupSolution**: Options at this level apply to the current project groupsolution. Diagram options apply to newly created diagrams within this... more (↗ see page 1088)
Together Option Categories (↗ see page 1088)	This section describes modeling option categories.
Option Value Editors (↗ see page 1101)	To edit an option, click the value field to invoke the appropriate value editor. There are several types of value editors: • **In-line text editor**. To edit a value in a text field, type the new value. Changes are applied when you press Enter or move the focus to another field. • **Combo box** or **list box**. Clicking a box field reveals the list of possible values. Select the required value from the list. • **Dialog box**. Clicking a dialog box field reveals the button that opens the dialog box. Specify the required values and click OK to apply changes.... more (↗ see page 1101)
Together Sequence Diagram Roundtrip Options (↗ see page 1102)	**Tools ▶ Options ▶ Together ▶ Various ▶ Roundtrip Options** Descriptions for sequence diagram roundtrip options. The Sequence Diagram Roundtrip options apply to generating sequence diagrams from source code and generating source code from a sequence diagram. The table below lists the Sequence Diagram Roundtrip options, descriptions, and default values.
Together Source Code Options (↗ see page 1103)	**Tools ▶ Options ▶ Together ▶ Various ▶ Source Code** Descriptions for source code options. The Source Code options allows you to control whether dependency links are drawn automatically on diagrams. The table below lists the Source Code general options, descriptions, and default values.
System macros (↗ see page 1103)	The following system macros can be used inside the text of some options: • **TIME**: current time • **LONGTIME**: current time (long format) • **DATE**: current date • **LONGDATE**: current date (long format) • **PROJECT**: project name • **DIAGRAM**: diagram name • **USER**: user name • **COMP**: computer name For example, if you use: `Project: %PROJECT%, diagram: %DIAGRAM%` It prints in the footer as: **Project: Project1, diagram: DgrClass1**

3.5.1.1 Configuration Levels

The configuration options apply to the four hierarchical levels. Each level contains a number of categories, or groups, of options.

The configuration options can be specified at the following levels:

- **Default**: Options at this level apply to all the current projects and project groupsolutions, as well as newly created ones. Diagram options apply to newly created diagrams within these projects and project groupsolutions. All options are available at this level.
- **Project groupSolution**: Options at this level apply to the current project groupsolution. Diagram options apply to newly created diagrams within this project groupsolution. All options are available at this level.
- **Project**: Options at this level apply to the current project. Diagram options apply to newly created diagrams within this project. All options except the Model View category are available at this level.
- **Diagram**: Options at this level apply to the current diagram. The Diagram options category is only available at this level.

See Also

Configuring Together (see page 183)

Options Dialog Window (see page 969)

3.5.1.2 Together Option Categories

This section describes modeling option categories.

Topics

Name	Description
Together Diagram Appearance Options (see page 1089)	**Tools ▶ Options ▶ Together ▶ Various ▶ Diagram ▶ Appearance** Descriptions for diagram appearance options. The Diagram options enable you to control a number of default behaviors and appearances of diagrams. The table below lists the Appearance options, descriptions, and default values.
Together Diagram Layout Options (see page 1091)	**Tools ▶ Options ▶ Together ▶ Various ▶ Diagram ▶ Layout** Descriptions for diagram layout options. Layout options define the alignment of diagram elements.
Together Diagram Print Options (see page 1094)	**Tools ▶ Options ▶ Together ▶ Various ▶ Diagram ▶ Print** Descriptions for diagram print options. The Print options define default settings that apply to your printed diagrams. Note that after these settings are applied to the printed material, OS-specific and printer-specific settings will be applied as well. The tables below list the Print options, descriptions, and default values.
Together Diagram View Filters Options (see page 1096)	**Tools ▶ Options ▶ Together ▶ Various ▶ Diagram ▶ View Management** Descriptions for view filters options. The View Filter group of options provide a set of filters that enable you to control the type of data displayed in different views of a model. The View Filters options control what elements display on your class and namespace (package) diagrams. The table below lists the filters, descriptions, and default values.
Together General Options (see page 1098)	**Tools ▶ Options ▶ Together ▶ Various ▶ General** Descriptions for general options. The General options allow you to customize certain behaviors in the user interface that do not pertain to any other specific category of options such as Diagram or View Filters. The table below lists the General options, descriptions, and default values.

Together Generate Documentation Options (↗ see page 1099)	**Tools ▶ Options ▶ Together ▶ Various ▶ Generate Documentation** Descriptions for generate documentation options. The Generate Documentation options control the variety of content (as well as appearance) to include or exclude from your generated HTML documentation. The table below lists the Generate Documentation options, descriptions, and default values.
Together Model View Options (↗ see page 1101)	**Tools ▶ Options ▶ Together ▶ Various ▶ Model View** Descriptions for Model View options. The Model View options control how diagram content displays in the **Model View.** The table below lists the Model View general options, descriptions, and default values.

3.5.1.2.1 Together Diagram Appearance Options

Tools ▶ Options ▶ Together ▶ Various ▶ Diagram ▶ Appearance

Descriptions for diagram appearance options.

The Diagram options enable you to control a number of default behaviors and appearances of diagrams. The table below lists the Appearance options, descriptions, and default values.

General group Options	Description and default value
Custom diagram background color	This parameter controls the background color of diagrams, if the Use default background color option is set to false. The default value is WhiteSmoke.
Diagram detail level	This option controls the amount of information displayed for an element in a diagram. The default value is Design. • **Design**: Names and types (visibility signs are shown). • **Analysis**: Names only (no visibility signs) • **Implementation**: Names and types, parameters for the methods, and initial values of attributes (visibility signs are shown).
Font in diagrams	This option defines the font and font size that is used in printed diagrams. The default value is Arial, 9.75pt.
Wrap text in nodes	This option controls whether the text displayed in a node automatically continues on the next line when it reaches the right border of the node. If set to False, the rest of the text is not displayed. The default value is True.
Maximum width of text labels (pixels)	This setting specifies a width limit for all text labels outside nodes (for example, link labels). The text automatically continues on the next line when it reaches this limit. If set to 0, the text is always displayed in a single line. The default value is 200.
Member format	This option controls the format of members in class diagrams. The default value is UML.
UML	Methods (functions) are displayed as `<name> (parameters) : <type>`. Fields are displayed as `<name> : type>`
Language	Methods in C# are displayed as `<type> <name>`. Fields in C# are displayed as `<type> <name>`. Functions in Visual Basic are displayed as `<name> (parameters) As <type>`. Fields in Visual Basic are displayed as `<name> As <type>`.

Show page borders	This option controls whether to show gray borders that represent page margins in the **Diagram View** and Overview. The default value is False.
Use default background color	This parameter controls whether the system background color is used in diagrams. If this parameter is true, then background color is defined according to the current Windows color scheme. If this parameter is false, the background color is defined by the Custom diagram background color parameter (see above). The default value is True.

Grid group Options	Description and default value
Grid color	This parameter defines the color of the diagram grid. The default value is LightGray.
Grid height (in pixels)	This option enables you to specify the exact height of grid squares in pixels. The default value is 10.
Grid style	This parameter controls whether the grid is displayed as dotted or solid lines. The default value is Lines.
Grid width (in pixels)	This option enables you to specify the exact width of grid squares in pixels. The default value is 10.
Show grid	If this option is true, a design grid is visible in the background behind diagrams. The default value is True.
Snap to grid	If this option is true, diagram elements "snap" to the nearest coordinate of the diagram background design grid. The snap function works whether the grid is visible or not. The default value is True.

Nodes group Options	Description and default value
3D look	If this option is true, a shadow appears under each diagram element to create a three-dimensional effect. The default value is True.
Show compartments as line	If this option is true, a control bar displays over the compartments of selected diagram elements (fields, methods, classes, and properties). The compartments are represented as expandable nodes that can be opened or closed by a mouse click. The default value is True.
Show imported classes with fully qualified names	This parameter controls whether imported class names are shown in the fully qualified or short form. The default value is True.
Show referenced class names	With this parameter, you can optionally show or hide the name of the base class or interface in the top-right corner of a classifier in the Diagram View. You can hide these references to simplify the visual presentation of the project. The default value is True.
Show referenced classes with fully qualified names	This parameter controls whether referenced class names are shown in the fully qualified or short form. The default value is True.

Sort according to positions in source code	Choose this option to sort fields, methods, subclasses and properties according to their positions in the source code. This option prevails over the others: if it is set to True, the options **Sort elements alphabetically** and **Sort elements by visibility** are ignored.
Sort elements alphabetically	This option controls the order of members displayed within elements on diagrams. If this option is true, fields, methods, subclasses, and properties are sorted alphabetically within compartments. The default value is True.
Sort elements by visibility	This option controls the order of members displayed within elements on diagrams. If this option is true, fields, methods, subclasses, and properties are sorted by visibility within compartments. The default value is True.

UML In Color group Options	**Description and default value**
Description stereotype	This parameter controls the color of classifiers with the stereotype "description." The default value is Light blue.
Mi-detail stereotype	This parameter controls the color of classifiers with the stereotype "Mi-detail." The default value is Light pink.
Moment-interval stereotype	This parameter controls the color of classifiers with the stereotype "moment-interval." The default value is Light pink.
Party stereotype	This parameter controls the color of classifiers with the stereotype "party." The default value is Light green.
Place stereotype	This parameter controls the color of classifiers with the stereotype "place." The default value is Light green.
Role stereotype	This parameter controls the color of classifiers with the stereotype "role." The default value is Yellow.
Thing stereotype	This parameter controls the color of classifiers with the stereotype "thing." The default value is Light green.
Enable UML in color	This option controls if the color of a classifier depends on the stereotype assigned. For each stereotype, you can select its individual color from the drop-down list (see above). The default value is True.

See Also

Configuring Together (see page 183)

Options dialog window (see page 969)

3.5.1.2.2 Together Diagram Layout Options

Tools ▶ Options ▶ Together ▶ Various ▶ Diagram ▶ Layout

Descriptions for diagram layout options.

Layout options define the alignment of diagram elements.

General group Options	Description and default value
Layout algorithm	UML diagrams can be thought of as graphs (with vertices and edges). Therefore, graph data structures (algorithms) can be applied to the UML diagrams for diagram layout. Click the drop-down arrow to select a layout algorithm. The various algorithms and their optional settings are described below. The algorithm that you specify executes when choosing **Layout ▶ Do full layout** on the diagram context menu. The following options are available: · <autoselect> · Hierarchical · Together · Tree · Orthogonal · Spring Embedder The default value is Together.
Recursive layout	This option is available for all layout algorithms. Selecting this option allows to layout all subelements within containers while laying out diagram nodes, thus enabling to lay out inner substructure. This option is useful for the composite states or components.

Hierarchical group Options	Description and default value
Hybrid proportion parameter	Used in conjunction with the Hybrid ordering heuristic. The optimal setting for this value is 0.7.
Inheritance	This option defines how nodes are aligned with each other if they are connected by an inheritance link. Horizontal - Nodes connected by inheritance are aligned horizontally Vertical - Nodes connected by inheritance are aligned vertically
Justification	This option defines the adjustment of nodes. The Justification setting depends on the Inheritance setting. Select from the following: · **Top:** If the Inheritance option is set as Vertical, all nodes in a column are aligned at the left of the column. If the Inheritance option is set as Horizontal, all nodes in a row are aligned at the top of the row. · **Center:** If the Inheritance option is set as Vertical, all nodes in a column are aligned at the center of the column. If the Inheritance option is set as Horizontal, all nodes in a row are aligned at the center of the row. · **Bottom:** If the Inheritance option is set as Vertical, all nodes in a column are aligned at the right of the column. If the Inheritance option is set as Horizontal, all nodes in a row are aligned at the bottom of the row.
Layer ordering heuristics	The heuristics are used to sort nodes among each layer to minimize edge-crossings: · **Barycenter:** The Barycenter heuristic reorders the nodes on node N according to the barycenter weight. The weight of node N is calculated as a simple average of all its successors/predecessors relative coordinates. · **Median:** The Median heuristic reorders the nodes on node N according to the median weight. The weight of node N is calculated as a simple average of the relative positions of this node dealing only with two central successors/predecessors coordinates. · **Hybrid:** The Hybrid heuristic combines the Median and Barycenter heuristics.
Minimal horizontal / vertical distance	Minimal allowed distance between elements in pixels. Here you can specify Vertical and Horizontal distance options.

Together group Options	Description and default value
Inheritance	This option defines how nodes are aligned with each other if they are connected by an inheritance link. Select either: · From left to right - nodes connected by inheritance are aligned horizontally from left to right. · From right to left - nodes connected by inheritance are aligned horizontally from right to left. · From top to bottom - nodes connected by inheritance are aligned vertically from top to bottom. · From bottom to top - nodes connected by inheritance are aligned vertically from bottom to top.
Justification	This option defines the adjustment of nodes. The Justification setting depends on the Inheritance setting. The elements are aligned as summarized in the following table: **Inheritance: Justification: Top:Center:Bottom:** Left-right Right of the columnCenter of the columnLeft of the column Right-leftLeft of the columnCenter of the columnRight of the column Top-bottomBottom of the rowCenter of the rowTop of the row Bottom-topTop of the rowCenter of the rowBottom of the row

Tree group Option	Description and default value
Hierarchy	This option defines the hierarchy direction of the elements. · Horizontal - Elements are aligned horizontally. · Vertical - Elements are aligned vertically.
Reverse hierarchy	The last in the hierarchy element is laid out first in the diagram.
Minimal horizontal (or: vertical) distance	Distance between elements is in pixels. Here you can specify Vertical and Horizontal distance options.
Justification	This option defines the adjustment of elements. The Justification setting is dependent on the Hierarchy direction setting. Select from the following: · Top: If the Hierarchy direction option is set to Vertical, all nodes in a column are aligned at the left of the column. If the Hierarchy direction option is set to Horizontal, all nodes in a row are aligned at the top of the row. · Center: If the Hierarchy direction option is set to Vertical, all nodes in a column are aligned at the center of the column. If the Hierarchy direction option is set to Horizontal, all nodes in a row are aligned at the center of the row. · Bottom:If the Hierarchy direction option is set to Vertical, all nodes in a column are aligned at the right of the column. If the Hierarchy direction option is set to Horizontal, all nodes in a row are aligned at the bottom of the row.
Process non-tree edges	If this option is selected, non-tree edges are bent to fit into the diagram layout.

Orthogonal group Options	Description and default value
Node placement strategy	There are three strategies for node placement: Tree, Balanced, and Smart. · Tree: The Tree node placement strategy creates a spanning-tree diagram layout. The spanning-tree for the given graph is calculated and diagram nodes are placed on the lattice to minimize the tree edges length. This minimizes the distance between nodes that are linked with a tree-edge. · Balanced:The Balanced node placement strategy uses a balanced ordering of the vertices of the graph as a starting point. Balanced means that the neighbors of each vertex V are as evenly distributed to the left and right of V as possible. · Smart:The Smart node placement strategy sorts all vertices according to the in/out degrees for each vertex and fills the lattice starting from the center with the vertices with the greatest degree.
Distance between elements	Distance is in pixels. Specifies the minimum distance between diagram elements.

Spring Embedder group Options	Description and default value
Spring force	Specify the rigidity of the springs. The greater value you specify, the less will be the length of edges in the final graph.
Spring movement factor	Specify the nodes movement factor. The more value you specify, the more distance will be between the nodes in the final graph. If you specify 0 as the movement factor, you will get random layout of the nodes.

See Also

Diagram Layout Overview (see page 92)

Configuring Together (see page 183)

Laying Out a Diagram (see page 202)

3.5.1.2.3 Together Diagram Print Options

Tools ▶ Options ▶ Together ▶ Various ▶ Diagram ▶ Print

Descriptions for diagram print options.

The Print options define default settings that apply to your printed diagrams.

Note that after these settings are applied to the printed material, OS-specific and printer-specific settings will be applied as well.

The tables below list the Print options, descriptions, and default values.

Appearance group Options	Description and default value
Print compartments as lines	When this option is true, a control bar displays over the compartments of diagram elements (fields, methods, classes, properties, and so on). The compartment nodes are expanded in the printed results. The default value is False.
Print shadows	When this option is true, a shadow appears under each diagram element in the printed results to create a three-dimensional effect. The default value is True.

Footer group Options	Description and default value
Footer alignment	This option enables you to select the footer alignment. The default value is Left.
Footer text	This option defines the text that is printed at the bottom of each page, if the Print footer option is true. System macros can be used in the text. See System macros (↗ see page 1103) The default value is `Printed by %USER% (%LONGDATE%)`.
Print footer	If this option is true, the text specified in the Footer text option is printed at the bottom of each page. The default value is True.

General print Options	Description and default value
Fit to page	If this option is true, the Print zoom setting is ignored, and the entire diagram prints on a single page. The default value is False.
Font	This option defines the font (and font size) used in printed diagrams. The default value is `Microsoft Sans Serif, 9.75pt`.
Print border	If this option is true, a border is printed around the edge of each page. The border corresponds to the Margins settings. The default value is True.
Print empty pages	If this option is true, printing includes any blank pages that appear. If this option is false, blank pages are skipped during printing. The default value is False.
Print zoom	This option defines the zoom factor for printing diagrams (1:1, 2:1, etc.). Think of the value 1 as being equal to 100%. A value of 2 prints a diagram at 200%, while 0.5 prints it at 50%, and so on. Use the decimal separator as defined in your regional settings. The default value is 1.

Header group Options	Description and default value
Header alignment	This option enables you to select the header alignment. The default value is Left.
Header text	This option defines the text that is printed at the top of each page, if the Print header option is true. System macros can be used in the text. See System macros (↗ see page 1103). The default value is `%PROJECT%, %DIAGRAM%`.
Print Header	If this option is true, the text specified in the Header text option is printed at the top of each page. The default value is True.

Margins group Options	Description and default value
Bottom margin	The bottom margin offset in inches. Use the decimal separator as defined in your regional settings. The default value is 1.
Left margin	The left margin offset in inches. Use the decimal separator as defined in your regional settings. The default value is 1.

Right margin	The right margin offset in inches. Use the decimal separator as defined in your regional settings. The default value is 1.
Top margin	The top margin offset in inches. Use the decimal separator as defined in your regional settings. The default value is 1.

Page numbers group Options	Description and default value
Page number alignment	This option enables you to select the page number alignment. The default value is Left.
Print page numbers	If this option is true, page numbers are printed. The default value is True.

Paper group Options	Description and default value
Custom paper height (in inches)	This option defines custom paper dimensions for printing. Settings here are only effective if the Paper size option is set to Custom. The default value is 11.88.
Custom paper width (in inches)	This option defines custom paper dimensions for printing. Settings here are only effective if the Paper size option is set to Custom. The default value is 8.4.
Paper orientation	This option defines the orientation of the page. If a Custom paper size is selected, Portrait orientation uses the width and height values specified in the Custom paper height and width settings, while Landscape orientation exchanges the width and height values. The default value is Portrait.
Paper size	This option defines the default paper dimensions for printing. The list of choices includes the most popular paper sizes. If you need to specify your own size, select Custom from the drop down list, and define the dimensions in the Custom paper height and Custom paper width fields. The default value is A4.

See Also

Configuring Together (see page 183)

Options dialog window (see page 969)

3.5.1.2.4 Together Diagram View Filters Options

Tools ▶ Options ▶ Together ▶ Various ▶ Diagram ▶ View Management

Descriptions for view filters options.

The View Filter group of options provide a set of filters that enable you to control the type of data displayed in different views of a model.

The View Filters options control what elements display on your class and namespace (package) diagrams. The table below lists the filters, descriptions, and default values.

Link filters group Options	Description and default value
Show association links	This filtering option controls showing/hiding association links in the current project. If it is set to true, association links are displayed. The default value is True.
Show dependency links	This filtering option controls showing/hiding dependency links in the current project. If it is set to true, dependency links are displayed. The default value is True.
Show generalization links	This filtering option controls showing/hiding generalization links in the current project. If it is set to true, generalization links are displayed. The default value is True.
Show implementation links	This filtering option controls showing/hiding implementation links in the current project. If it is set to true, implementation links are displayed. The default value is True.

Member filters group Options	Description and default value
Show fields	This filtering option controls showing/hiding fields in the current project. If it is set to true, fields are displayed. The default value is True.
Show indexers	This filtering option controls showing/hiding indexers in the current project. If it is set to true, indexers are displayed. The default value is True.
Show members	This filtering option controls showing/hiding members in the current project. If it is set to true, members are displayed. The default value is True.
Show methods	This filtering option controls showing/hiding methods in the current project. If it is set to true, methods are displayed. The default value is True.
Show non public members	This filtering option controls showing/hiding nonpublic members in the current project. If it is set to true, nonpublic members are displayed. The default value is True.
Show properties	This filtering option controls showing/hiding properties in the current project. If it is set to true, properties are displayed. The default value is True.

Node filters group Options	Description and default value
Show classes	This filtering option controls showing/hiding classes in the current project. If it is set to true, classes are displayed. The default value is True.
Show constraints	This filtering option controls showing/hiding OCL constraints in the current project. If it is set to true, constraints are displayed. The default value is True.

Show delegates	This filtering option controls showing/hiding delegates in the current project. If it is set to true, delegates are displayed. The default value is True.
Show enumerations	This filtering option controls showing/hiding enums in the current project. If it is set to true, enums are displayed. The default value is True.
Show events	This filtering option controls showing/hiding events in the current project. If it is set to true, events are displayed. The default value is True.
Show interfaces	This filtering option controls showing/hiding interfaces in the current project. If it is set to true, interfaces are displayed. The default value is True.
Show modules	This filtering option controls showing/hiding modules in the current project. If it is set to true, modules are displayed. The default value is True.
Show namespaces	This filtering option controls showing/hiding namespaces in the current project. If it is set to true, namespaces are displayed. The default value is True.
Show non public classes	This filtering option controls showing/hiding nonpublic classes in the current project. If it is set to true, nonpublic classes are displayed. The default value is True.
Show notes	This filtering option controls showing/hiding notes in the current project. If it is set to true, notes are displayed. The default value is True.
Show shortcuts	This filtering option controls showing/hiding shortcuts in the current project. If it is set to true, shortcuts are displayed. The default value is True.
Show structures	This filtering option controls showing/hiding structures in the current project. If it is set to true, structures are displayed. The default value is True.

See Also

Configuring Together (see page 183)

Options dialog window (see page 969)

3.5.1.2.5 Together General Options

Tools ▶ Options ▶ Together ▶ Various ▶ General

Descriptions for general options.

The General options allow you to customize certain behaviors in the user interface that do not pertain to any other specific category of options such as Diagram or View Filters. The table below lists the General options, descriptions, and default values.

General Group Options	Description and default value
Automatically reopen diagrams	This option controls whether diagrams are reopened automatically when you reopen a project. If this option is true, Together reopens all diagrams that were open at the time when you closed the project. If this option is false, diagrams are not reopened, which helps to decrease project opening time. The default value is True.
Delete Confirmation	This option defines whether confirmation is requested before deleting a namespace, classifier, or diagram. The default value is True.

Together Support Group Options	Description and default value
Automatically enable Together support for new and opened projects	This option defines whether Together support is automatically enabled for opened and new projects added to an existing project groupsolution. A project is considered new when it is created within an existing project groupsolution using File | New | Project. Note that Together support, enabled using the Model Support dialog, overrides this setting. The default value is True.
Automatically enable Together support for opened projects	This option defines whether Together support is automatically enabled for projects opened within a project groupsolution that have never been exposed to modeling support. When a project groupsolution is created, its default project is regarded as opened. Note that Together support, enabled using the **Model Support** dialog box, overrides this setting. The default value is True.
Short name of the folder where model support files are stored	This option allows you to specify the name of the project subfolder where all model files are stored. The option is useful when you import or share projects created in other editions of CodeGear Together. In order to make effect, this option needs to be adjusted before creating a project. The default value is `ModelSupport_%PROJECTNAME%ModelSupport`

See Also

Interoperability Overview

Configuring Together (see page 183)

Activating Together Support for Projects (see page 263)

3.5.1.2.6 Together Generate Documentation Options

Tools ▶ Options ▶ Together ▶ Various ▶ Generate Documentation

Descriptions for generate documentation options.

The Generate Documentation options control the variety of content (as well as appearance) to include or exclude from your generated HTML documentation. The table below lists the Generate Documentation options, descriptions, and default values.

General group Options	Description and default value
Bottom	Specifies the text to be placed at the bottom of each output file. The text will be placed at the bottom of the page, below the lower navigation bar. The text can contain HTML tags and white spaces.

Documentation Title	Specifies the title to be placed at the top of the overview summary file. The title will be placed as a centered, level-one heading directly beneath the upper navigation bar. It can contain HTML tags and white spaces.
Footer	Specifies the footer text to be placed at the bottom of each output file. The footer is placed to the right of the lower navigation bar. It can contain HTML tags and white spaces.
Header	Specifies the header text to be placed at the top of each output file. The header is placed to the right of the upper navigation bar. It can contain HTML tags and white spaces.
Navigation type	Specifies the location for descriptions of design elements: Package: in package files Diagram: in diagram files The default value is Package.
Use Internal Browser	When this options is true, an internal browser is used for documentation presentation. The default value is False.
Window Title	Specifies the title to be placed in the HTML `<title>` tag. This text appears in the window title and in any browser bookmarks (favorites) created for this page. This title should not contain any HTML tags, since the browser does not interpret them properly.

Include group Options	Description and default value
internal (package)	If this option is true, internal classes (friend in VB), interfaces and members are shown in the generated documentation. The default value is True.
private	If this option is true, private classes, interfaces and members are shown in the generated documentation. The default value is False.
protected	If this option is true, protected classes, interfaces and members are shown in the generated documentation. The default value is False.
protected internal	If this option is true, protected internal classes (protected friend in VB), interfaces and members are shown in the generated documentation. The default value is False.
public	If this option is true, public classes, interfaces and members are shown in the generated documentation. The default value is True.

Navigation group Options	Description and default value
Generate Help	This option controls whether to put the HELP link in the navigation bars at the top and bottom of each page of output. The default value is True.
Generate Index	This option controls whether to generate the index. The default value is True.
Generate Navbar	This option controls whether to generate the navigation bar, header, and footer, otherwise found at the top and bottom of the generated pages. The default value is True.

Generate Tree	This option controls whether to generate the class/interface hierarchy. The default value is True.
Generate Use	If this option is true, one Use page is included for each documented class and namespace. The page describes which namespaces, classes, methods, constructors, and fields use any API of the given class or namespace. Given class C, things that use class C would include subclasses of C, fields declared as C, methods that return C, and methods and constructors with parameters of type C. The default value is False.

See Also

Configuring Together (see page 183)

Options dialog window (see page 969)

3.5.1.2.7 Together Model View Options

Tools ▶ Options ▶ Together ▶ Various ▶ Model View

Descriptions for Model View options.

The Model View options control how diagram content displays in the **Model View.** The table below lists the Model View general options, descriptions, and default values.

Option	Description and default value
Show diagram nodes expandable	If this option is set to true, the Model View displays expandable diagram nodes with the elements contained therein. The default value is True.
Show links	If this option is set to true, the Model View displays links between nodes. Otherwise the links are hidden. The default value is False.
Sorting type	This option enables you to select the type of sorting for the Model View (alphabetical, by metaclass, or none). The default value is Metaclass.
View type	This option controls the type of presentation of the Model. Diagram-centric mode assumes that the design elements are shown in their respective diagrams, and only the classes, interfaces and the other source-code elements are shown in the namespaces. Model-centric mode assumes that all elements are shown under namespaces. The default value is Diagram-centric.

See Also

Configuring Together (see page 183)

Options dialog window (see page 969)

3.5.1.3 Option Value Editors

To edit an option, click the value field to invoke the appropriate value editor. There are several types of value editors:

- **In-line text editor**. To edit a value in a text field, type the new value. Changes are applied when you press Enter or move the focus to another field.
- **Combo box** or **list box**. Clicking a box field reveals the list of possible values. Select the required value from the list.

- **Dialog box**. Clicking a dialog box field reveals the button that opens the dialog box. Specify the required values and click OK to apply changes.

See Also

Configuring Together (see page 183)

Options Dialog Window (see page 969)

3.5.1.4 Together Sequence Diagram Roundtrip Options

Tools ▶ Options ▶ Together ▶ Various ▶ Roundtrip Options

Descriptions for sequence diagram roundtrip options.

The Sequence Diagram Roundtrip options apply to generating sequence diagrams from source code and generating source code from a sequence diagram. The table below lists the Sequence Diagram Roundtrip options, descriptions, and default values.

Option	Description and default value
General	
Show full diagnostic	This option specifies whether to show diagnostic messages produced when generating sequence diagrams. If this option is True, the parser can report detailed progress as it runs. The default value is True.
Generate sequence diagram	
Depth of call nesting	During sequence diagram generation, this value limits how deep the parser traverses the source code calling sequence. This value can help keep the generated sequence diagram from becoming so large that it is unusable. You may want to use this option as a means for quickly generating a high-level sequence diagram for a complex method by using a low call nesting value (1-3). Based on the results of your diagram, you can choose to look deeper and/or create additional sequence diagrams for each of the major methods uncovered. The default value is 10.
Include messages to self	This option specifies whether to show messages to self on the generated sequence diagram. If this option is True, messages to self are shown. The default value is True.
Generate source code	
Never change existing code	If this option is True, existing code can never be changed when generating source code from a sequence diagram. If this option is False and some source code already exists, a confirmation dialog opens when you generate source code from a sequence diagram, and you can choose whether to modify the existing code. The default value is True.

See Also

Roundtrip engineering overview

Configuring Together (see page 183)

Options dialog window (see page 969)

3.5.1.5 Together Source Code Options

Tools ▶ Options ▶ Together ▶ Various ▶ Source Code

Descriptions for source code options.

The Source Code options allows you to control whether dependency links are drawn automatically on diagrams. The table below lists the Source Code general options, descriptions, and default values.

Option	Description and default value
Autocreate association links derived from properties	Set this option to True to create association links for properties automatically, while parsing the source code. The default value is False.
Autocreate dependency links	If the option is true, the dependency links are drawn automatically on diagrams. The default value is False.
Automatically maintain namespace folder structure	If this parameter is true, new classes will be created in subfolders. The structure of these subfolders will be detected using the existing folder structure, if any. For new namespaces, new subfolders will be created. If this parameter is false, new classes will be created in the project root folder. The default value is True.
Encoding	This parameter specifies the character encoding to be applied to source code files. System default lets you use the current OS encoding. The default value is system default.
Use cache	Cache can be used for implementation projects. With this cache, reopening a project or a diagram becomes much faster. The default value is True.

See Also

Configuring Together (see page 183)

3.5.1.6 System macros

The following system macros can be used inside the text of some options:

- **TIME**: current time
- **LONGTIME**: current time (long format)
- **DATE**: current date
- **LONGDATE**: current date (long format)
- **PROJECT**: project name
- **DIAGRAM**: diagram name
- **USER**: user name
- **COMP**: computer name

For example, if you use: `Project: %PROJECT%, diagram: %DIAGRAM%`

It prints in the footer as: **Project: Project1, diagram: DgrClass1**

See Also

Configuring Together (see page 183)

3.5.2 Together Keyboard Shortcuts

Together enables you to perform many diagram actions without using the mouse. You can navigate between diagrams, create diagram elements, and more, using the keyboard only.

Navigational shortcut keys

Keyboard shortcuts for navigation and browsing:

Action	Shortcut	Notes
Navigate between open diagrams in the **Diagram View**	CTRL+Tab	The title of the diagram that has focus is in bold text.
Expand node in **Model View**	Right arrow	
Collapse node in **Model View**	Left arrow	
Open the Object InspectorProperties Window	F4, or Alt + Enter	
Close current diagram	CTRL+F4	
Toggle between a selected container node and its members	PgDown/PgUp	
Navigate between nodes or node members	Arrow keys, Shift + arrow keys	

Shortcut keys for editing

Keyboard shortcuts for editing:

Action	Shortcut
Cut, Copy, or Paste model elements or members.	CTRL+X, CTRL+C, CTRL+V
Activate the in-place editor for a diagram element to edit, rename a member.	F2
Undo	CTRL+Z
Redo	CTRL+Y, CTRL+SHIFT+Z
Select all elements on the diagram	CTRL+A
Close the Overview window	ESC
Add a new namespace (package) to a diagram	CTRL+E
Add a new class to a diagram	CTRL+L
Add new method (operation) to a class or interface	CTRL+M
Add a new field (attribute) to a class	CTRL+W
Add a new interface to diagram	CTRL+SHIFT+L
Open the Add Shortcuts dialog box	CTRL+SHIFT+M
Add a new diagram from the **Model View**	CTRL+SHIFT+D

Zoom shortcut keys

Keyboard shortcuts for zooming the diagram image:

Action	Shortcut	Notes
Zoom in	+	Use the numeric keypad
Zoom out	-	Use the numeric keypad
Fit the entire diagram in the **Diagram View**	*	Use the numeric keypad
Display the actual size	/	Use the numeric keypad

Other shortcut keys

Other keyboard shortcuts:

Action	Shortcut
Open the Print Diagram dialog box	CTRL+P
Diagram update	F6

See Also

Help on Help (see page 51)

About Together (see page 83)

3.5.3 GUI Components for Modeling

This section describes GUI components of the RAD Studio interface you use for UML modeling.

Topics

Name	Description
Diagram View (see page 1106)	**Context menu (in the Model View)** ▶ **Open Diagram** The **Diagram View** displays model diagrams. Each diagram is presented in its own tab. To open the **Diagram View**, choose a diagram, namespace or a package in the **Model View**, right-click it and choose Open Diagram on the context menu. Most manipulations with diagram elements and links involve drag-and-drop operations or executing right-click (or context) menu commands on the selected elements. Some of the actions provided by the context menus are: • Add or delete diagram elements and links • Add or delete members in the elements • Create elements by pattern • Cut,... more (see page 1106)
Pattern GUI Components (see page 1107)	This section describes GUI components of the RAD Studio interface you use for Together Pattern features.
Menus (see page 1110)	
Quality Assurance GUI Components (see page 1110)	This section describes GUI components of the RAD Studio interface you use for Together Quality Assurance features.

Model View (see page 1112)	**View ▶ Model View** The **Model View** provides the logical representation of the model of your project: namespaces (packages) and diagram nodes. Using this view, you can add new elements to the model; cut, copy, paste and delete elements, and more. Context menu commands of the **Model View** are specific to each node. Explore these commands as you encounter them. In the **Model View**, only the nodes and their respective subnodes shown in the **Diagram View** are listed under the corresponding diagram node. For example, if you have a namespace (package) containing a class, both the namespace (package) and class... more (see page 1112)
Object Inspector (see page 1113)	View->Object Inspector View->Properties Window When Together support is activated, the Object InspectorProperties Window shows the properties of an element that is selected in the Model or Diagram Views. To view the Object InspectorProperties Window, choose Object InspectorProperties Window from the View menu, press `F4`, or press `ALT+ENTER`. The content of the Object InspectorProperties Window depends on the element type. You can use the Object InspectorProperties Window to edit diagram or element properties. There are several categories of properties:
Tool Palette (see page 1114)	View->Tool Palette View->Toolbox Together extends the Tool PaletteToolbox of RAD Studio by adding model elements to it. The RAD Studio Tool PaletteToolbox displays special tabs for the supported UML diagrams. When a diagram is opened in the **Diagram View**, the appropriate tab appears in the Tool PaletteToolbox. In the Tool PaletteToolbox you see model elements (nodes, links) that can be placed on the current diagram. However, you can choose to show the tabs for all diagram types. Use the Tool PaletteToolbox buttons to create the diagram contents. **Note:** The set of available model... more (see page 1114)

3.5.3.1 Diagram View

Context menu (in the Model View) ▶ Open Diagram

The **Diagram View** displays model diagrams. Each diagram is presented in its own tab.

To open the **Diagram View**, choose a diagram, namespace or a package in the **Model View**, right-click it and choose Open Diagram on the context menu.

Most manipulations with diagram elements and links involve drag-and-drop operations or executing right-click (or context) menu commands on the selected elements.

Some of the actions provided by the context menus are:

- Add or delete diagram elements and links
- Add or delete members in the elements
- Create elements by pattern
- Cut, copy, and paste the selected items
- Navigate to the source code
- Hyperlink diagrams
- Zoom in and out

Item	Description
Working area	The main part of the **Diagram View** shows the current diagram.

Context menu	The context menus of the **Diagram View** are context-sensitive. Right-clicking model elements, including class members, provides access to element-specific operations on the respective context menu. Right-clicking the diagram background opens the context menu for the diagram.
Overview button	Opens the Overview pane (see below).

Overview pane

The overview feature of the **Diagram View** provides a thumbnail view of the current diagram. The Overview button is located in the bottom right corner of every diagram.

OCL Editor

The OCL Editor is used to enter and edit OCL expressions. Any changes to the names of your model components (classes, operations, attributes, and so on) used in these expressions are automatically updated by Together. This guarantees that your OCL constraints always stay up-to-date.

See Also

Diagram Overview (↗ see page 90)

OCL Support Overview (↗ see page 95)

Creating a Diagram (↗ see page 196)

Synchronizing Model View (↗ see page 267)

3.5.3.2 Pattern GUI Components

This section describes GUI components of the RAD Studio interface you use for Together Pattern features.

Topics

Name	Description
Pattern Organizer (↗ see page 1107)	**Tools ▶ Pattern Organizer** The Pattern Organizer window enables you to logically organize the patterns found in the Pattern Wizard using virtual trees, folders and shortcuts. You can also view and edit pattern properties.
Pattern Registry (↗ see page 1109)	**Pattern Organizer context menu ▶ New Shortcut (or: Assign Pattern)** The **Pattern Registry** defines the virtual hierarchy of patterns. When you create a folder or a shortcut in the Pattern Organizer and save the changes, a new entry is added to the registry of shortcuts. All operations with the contents of the Pattern Registry are performed in the **Pattern Organizer**, and synchronized with the registry. The window opens when using New Shortcut and Assign Pattern commands on the **Pattern Organizer** context menu.

3.5.3.2.1 Pattern Organizer

Tools ▶ Pattern Organizer

The Pattern Organizer window enables you to logically organize the patterns found in the Pattern Wizard using virtual trees, folders and shortcuts. You can also view and edit pattern properties.

Virtual pattern trees

This section displays the logical hierarchy of patterns. Context menus are provided for the root folder node, subfolders, and the pattern elements. The root folder context menu items are as follows:

Menu item	Description
New Pattern Tree	Use this command to create a new Pattern Tree node.
Sort Folder	Sorts nodes in ascending alphabetical order.

The subfolders beneath the root folder contain the following context menu items:

Menu item	Description
New Folder	Use this command to create a new subfolder under the selected folder.
New Shortcut	Opens the **Pattern Registry** allowing you to create a new shortcut to a pattern. The shortcut is placed in the selected folder.
Cut	Cuts the selected node to the clipboard.
Copy	Copies the selected node to the clipboard.
Paste	Pastes the clipboard contents to the selected node.
Delete	Removes the selected node.
Sort Folder	Sorts nodes in ascending alphabetical order.

The context menu for pattern elements contains the following menu items:

Assign Pattern	Opens the **Pattern Registry** allowing you to assign a pattern to the selected pattern element.
Cut	Cuts the selected node to the clipboard.
Copy	Copies the selected node to the clipboard.
Paste	Pastes the clipboard contents to the selected node.
Delete	Removes the selected node.

Properties

This section displays properties of the selected pattern or folder. You can edit the Name and Visible fields.

Name	The name displayed in the Virtual pattern tree. This field is editable.
Valid	This field applies only to patterns. If you have registered the pattern using the **Pattern Registry**, then the status is reported as valid. Otherwise, the pattern status is Invalid, and it will not display in the **Pattern Wizard**. Folders are always considered valid and shown in the Pattern Wizard dialog (unless hidden using the Visible property).
Visible	Use the combobox to specify whether the pattern or folder is visible or hidden in the Pattern Wizard dialog.

Pattern description

A read-only section that displays comments for the selected pattern.

Edit Shared Patterns Root

Click this button to open the list of shared patterns roots. Use **Add** and **Remove** buttons to make up the desired list, and click **OK** when ready.

See Also

Patterns Overview (see page 96)

Pattern Registry (see page 1109)

Pattern Wizard (see page 1162)

3.5.3.2.2 Pattern Registry

Pattern Organizer context menu ▶ New Shortcut (or: Assign Pattern)

The **Pattern Registry** defines the virtual hierarchy of patterns. When you create a folder or a shortcut in the Pattern Organizer and save the changes, a new entry is added to the registry of shortcuts. All operations with the contents of the Pattern Registry are performed in the **Pattern Organizer**, and synchronized with the registry.

The window opens when using New Shortcut and Assign Pattern commands on the **Pattern Organizer** context menu.

Filters

The filters determine which patterns to display in the Patterns table. The Filters section provides the following set of filtering options:

Option	Description
Category	The available options are: Class Link Member Other All
Register	This option filters patterns by status of registration: All - all existing patterns Already - the patterns assigned to shortcuts None - the patterns not assigned to shortcuts
Containers	This option filters patterns by the container metaclass.
Diagram type	This option filters patterns that pertain to the selected diagram type.
Language	This option is available for implementation projects only, and enables you to filter out language-specific patterns.

Main sections

Option	Description
Patterns	Displays a table with the names and types of available patterns.
Pattern Properties	Displays the properties of the pattern selected in the Patterns table.
Pattern Description	A read-only section that displays comments for the pattern selected in the Patterns table.

Buttons

Button	Description
Synchronize	Click this button to search for patterns throughout your storage and update the **Pattern Registry**.
OK	Click this button to save the filtering settings and close the window.
Cancel	Click this button to discard the filtering settings and close the window.
Help	Displays this page.

See Also

Patterns Overview (see page 96)

Pattern Organizer (see page 1107)

Pattern Wizard (see page 1162)

3.5.3.3 Menus

Item	Description
File menu	You can use the File menu to export diagrams to image files, and print diagrams.
Edit menu	Use the Edit menu to cut, copy, paste, and delete diagrams and diagram elements, select all items on a diagram, and undo/redo actions.
View menu	The View menu contains the command for opening the Model View.
Project menu	Use the Project menu to enable or disable Together support for specific projects in the project groupsolution currently open.
RefactorRefactoring menu	The RefactorRefactoring menu contains refactoring commands for the implementation projects.
Tools menu	Use the Tools menu to generate documentation, open the pattern registry and pattern organizer, run audits and metrics (for the implementation projects), and set Together-specific options.
Diagram context menu	Use the Diagram context menu to add new elements, manage the layout, zoom in and out, show or hide diagram elements, synchronize your diagram with the Model view, and edit hyperlinks.
Model View context menu	Context menus of the various elements of the Model View are context-sensitive. The list of elements that can be added to a diagram from the Model View depends on the element and diagram type.
Element context menus	You can add or delete members (or delete the element itself), cut/copy/paste, view source code, show element information and more. Explore the context menus of the different elements as you encounter them to see what is available for each one.

See Also

Model View (↗ see page 1112)

Diagram View (↗ see page 1106)

3.5.3.4 Quality Assurance GUI Components

This section describes GUI components of the RAD Studio interface you use for Together Quality Assurance features.

Topics

Name	Description
Audit Results Pane (↗ see page 1111)	**QA Audits dialog window ▶ Start button** Use this pane to view and export audit results. Audit results are displayed as a table in the Audits View. Each time that you generate audits for the same project, the audit results display in a tabbed-page format with the most recent results having focus in the window. Although the audit results open initially as a free-floating window, it is a dockable window. The docking areas are any of the four borders of the RAD Studio window. You can position the audit results window according to your preferences. **Tip:** Press F1 with the Audits... more (↗ see page 1111)
Metric Results Pane (↗ see page 1112)	**QA Metrics dialog window ▶ Start button** Use this pane to view and export metric results. The metrics results report is displayed as a table in the **Metrics results pane**. The rows display the elements that were analyzed, and the columns display the corresponding values of selected metrics. Context menus of the rows and columns enable you to navigate to the source code, view descriptions of the metrics, and produce graphical output. **Tip:** Press F1 with the **Metrics results pane** having the focus to display this page.

3.5.3.4.1 Audit Results Pane

QA Audits dialog window ▶ Start button

Use this pane to view and export audit results.

Audit results are displayed as a table in the Audits View.

Each time that you generate audits for the same project, the audit results display in a tabbed-page format with the most recent results having focus in the window.

Although the audit results open initially as a free-floating window, it is a dockable window. The docking areas are any of the four borders of the RAD Studio window. You can position the audit results window according to your preferences.

Tip: Press F1

with the Audits View having the focus to display this page.

Item	Description
Toolbar buttons	
Save Audit results	Saves audit results.
Print Audit results	Prints audit results.
Refresh	Recalculates the results that are currently displayed.
Restart	Opens the **Audits** dialog window, define new settings and start new audits analysis.
Context menu commands	
Group By	Groups items according to the selected column.
Show Description	Displays a window with the full name and description of the selected audit.
Open	Opens the selected element in the source code editor highlighting the relevant code.
Copy	You can copy one row or multiple rows in the Audit results. Use `CTRL+CLICK` to select multiple rows to copy.
Close	Closes the current tab.
Close All	Closes all tabs and the results window.
Close All But This	Closes all tabs except for the tab currently in focus.

Note that only audit violations are shown in the results table. For this reason, the results do not necessarily display all of the audits that you ran, or all the packages or classes that you processed. The table contains the following columns:

Audit violations

Violation table column	Description
Abbreviation	The abbreviation for the audit name. The full name is displayed in the description (choose Show Description on the context menu of the violation).
Description	Describes why the audit flagged the item.
Severity	Indicates how serious, in general, violations of the audit are considered to be. This will help you sort the results and assess which violations are critical and which are not.
Resource	The source code item that was flagged by the audit.
File	The file that contains the problem code.

Line	The line number in the file where the problem code is located.

See Also

Quality Assurance Facilities Overview (see page 98)

Viewing Audit Results (see page 274)

Running Audits (see page 273)

3.5.3.4.2 Metric Results Pane

QA Metrics dialog window ▶ Start button

Use this pane to view and export metric results.

The metrics results report is displayed as a table in the **Metrics results pane**. The rows display the elements that were analyzed, and the columns display the corresponding values of selected metrics. Context menus of the rows and columns enable you to navigate to the source code, view descriptions of the metrics, and produce graphical output.

Tip: Press F1

with the **Metrics results pane** having the focus to display this page.

Item	Description
Toolbar buttons	
Save	Saves metric results.
Refresh	Recalculates the results that are currently displayed.
Restart	Opens the Metrics dialog window, define new settings and start new metrics analysis.
Context menu commands	
Show Description	Displays a window with the full name and description of the selected metric.
Chart	Builds a metric chart.

See Also

Quality Assurance Facilities Overview (see page 98)

Running Metrics (see page 276)

Viewing Metric Results (see page 276)

3.5.3.5 Model View

View ▶ Model View

The **Model View** provides the logical representation of the model of your project: namespaces (packages) and diagram nodes. Using this view, you can add new elements to the model; cut, copy, paste and delete elements, and more. Context menu commands of the **Model View** are specific to each node. Explore these commands as you encounter them.

In the **Model View**, only the nodes and their respective subnodes shown in the **Diagram View** are listed under the corresponding diagram node. For example, if you have a namespace (package) containing a class, both the namespace (package) and class are shown under the diagram node in the **Model View**. However, any members of the class are not shown under the diagram node as they are displayed under the namespace (package) node only.

Although the **Model View** opens initially as a free-floating window, it is a dockable window. The docking areas are any of the four borders of the RAD Studio window. You can position the **Model View** window according to your preferences.

The following options are applicable to the **Model View**:

- In the **Show diagram nodes expandable** field (**Options ▶ Together ▶ Model View**) choose *True* to show, or *False* to hide expandable diagram nodes. By default, the **Model View** displays expandable diagram nodes with the elements contained therein. You can hide expandable diagram nodes to further simplify the visual presentation of the project.
- In the **Show links** field (**Options ▶ Together ▶ Model View**) choose *True* to show, or *False* to hide expandable diagram nodes. By default, the Model View does not display links between nodes. You can opt to show the links to make the visual presentation of the project more detailed.
- In the **Sorting type** field (**Options ▶ Together ▶ Model View**) choose *Metaclass*, *Alphabetical*, or *None* to sort elements in the **Model View**. By default, diagram nodes are sorted by metaclass. You can sort elements in the **Model View** by metaclass, alphabetically, or none.
- In the **View type** field (**Options ▶ Together ▶ Model View**) choose Diagram-centric or Model-centic from the list box. For the sake of better presentation you can opt to show your model in diagram-centric or in model-centric modes. The Diagram-centic mode assumes that the design elements are shown under their respective diagrams; the namespaces only contain classes and interfaces (and the source-code elements for implementation projects). The Model-centric mode assumes that all elements are shown under the namespaces.

Item	Description
Root project node	The topmost item of a project structure.
Nodes	Namespaces (packages), diagrams, then model elements of the current model.
Refresh Model View button	Updates the model structure in the Model View to show possible changes in source code.
Regenerate ECO source code button	
Update ECO source code button	

Reload command

This command is available for implementation projects only.

You can use the Reload command (available at the root project node) to refresh the Together model from the source code. This command provides a total refresh for the elements and removes invalid code elements from the model. Using this command has the same effect as reopening the project groupsolution, but avoids the overhead of reinitializing RAD Studio.

See Also

Modeling Overview (see page 89)

Creating a Diagram (see page 196)

Synchronizing Model View (see page 267)

Troubleshooting a Model (see page 269)

Transforming Design Project to Source Code (see page 269)

3.5.3.6 Object Inspector

View->Object Inspector View->Properties Window

When Together support is activated, the Object InspectorProperties Window shows the properties of an element that is selected in the Model or Diagram Views. To view the Object InspectorProperties Window, choose Object InspectorProperties Window from the View menu, press `F4`, or press `ALT+ENTER`. The content of the Object InspectorProperties Window depends on the element type.

You can use the Object InspectorProperties Window to edit diagram or element properties.

There are several categories of properties:

Item	Description
Description	Text editor field where you can optionally provide textual description of an element.
Design	These properties are used to define the appearance of an element.
General	UML and source code element properties.
User properties	This node appears when there are user properties defined. In addition to the standard properties, you can define an unlimited number of user-defined properties.

Using the Object InspectorProperties Window, you can view and edit the properties of an element. Clicking on an editable field reveals which type of internal editor is available: text area, combobox with a list of values, or a dialog box. The read-only fields are displayed gray. As you click on the element properties, their respective descriptions display at the bottom of the Object InspectorProperties Window.

See Also

Working with User Properties (↗ see page 206)

Option Value Editors (↗ see page 1101)

3.5.3.7 Tool Palette

View->Tool Palette View->Toolbox

Together extends the Tool PaletteToolbox of RAD Studio by adding model elements to it.

The RAD Studio Tool PaletteToolbox displays special tabs for the supported UML diagrams. When a diagram is opened in the **Diagram View**, the appropriate tab appears in the Tool PaletteToolbox.

In the Tool PaletteToolbox you see model elements (nodes, links) that can be placed on the current diagram. However, you can choose to show the tabs for all diagram types. Use the Tool PaletteToolbox buttons to create the diagram contents.

Note: The set of available model elements depends on the type of a diagram that is currently selected in the Diagram View

. For descriptions of available elements, refer to Together Reference.

Tip: You can control the diagram elements that appear in the Tool Palette

Toolbox and create your own Tool PaletteToolbox tabs with specified items. You can cut, copy, paste, delete, rename, move up, and move down Tool PaletteToolbox items to customize your Tool PaletteToolbox view. You can also display Tool PaletteToolbox items alphabetically or in a list. Use the Tool PaletteToolbox context menu to accomplish such tasks. For further information, please refer to the RAD Studio documentation.

See Also

Creating a Single Element (↗ see page 209)

Creating a Simple Link (↗ see page 209)

3.5.4 Together Refactoring Operations

The following refactoring operations are available in Together:

Refactoring operations

Operation	Description
Change parameters	You can rename, add, or remove parameters for a single method using the **Model View**, **Diagram View**, or the Editor.
Extract interface	The Extract Interface command creates a new interface from one or more selected classes. Each selected class should implement the interface.
Extract method	You can extract method from a class or interface using the Editor only. It is important that the code fragment to be extracted must contain complete statements.
Extract superclass	The Extract Superclass command creates an ancestor class from several members of a given class.
Inline variable	If you have a temporary variable that is assigned once with a simple expression, you can replace all references to that variable with the expression using the Inline Variable command in the RAD Studio Editor.
Introduce field	Creates a new field.
Introduce variable	Creates a new variable. This command is available in the Editor.
Move members	Moving members only applies to the static methods, static fields and static properties (static members). This command is available on the Diagram View, on the **Model View**, and in the Editor.
Pull Members Up	Use this command to copy a member (field, method, property, indexer, and event) from a subclass to a superclass, optionally making it abstract. If there are no superclasses, an error message is displayed.
Push Members Down	Use this command to copy a member (field, method, property, and event) from a superclass to a subclass, optionally making it abstract. If there are no subclasses, an error message is displayed. Indexers cannot be pushed down.
Rename	You can rename certain code elements: class, interface, method, field, parameter, local variable, and so on. Together propagates the name changes to the dependent code in your project files. You cannot rename indexers, constructors, or destructors. You can rename namespaces by changing their names in the Diagram or Model Views. Together makes the appropriate changes in this case as well. This action does not require you to use the Refactoring commands. When renaming a method, all descendant classes are scanned for possible overrides. All found overrides and their usages are renamed too. However, if the method being renamed overrides some base class method, you are asked if you want to rename this base class method (and all overrides as well), or only the selected method (and its overrides). Refactoring also checks whether the method name that you enter coincides with some base class method. In such cases, a warning is displayed. When renaming a method that belongs to a child class, you can choose to propagate renaming to the parent classes and overloads, checking the respective options Refactor Ancestors and **Rename Overloads** in the **Rename Method** dialog. If such renaming in the ancestor classes is impossible, a warning message appears. However, you can still rename the selected method only. If you rename a model element that is referenced from another project, RAD Studio updates both projects.
Safe Delete	The Safe Delete command searches your code for any usages of the element that you wish to delete. You can invoke the command from the **Diagram View**, **Model View**, or from the Editor.

See Also

Refactoring Overview (see page 98)

Using Refactor Operations (↗ see page 184)

3.5.5 Project Types and Formats with Support for Modeling

There are two basic project types:

- **Design project.** Project file extension: `.bdsproj.tgproj`. These projects are language-neutral and comply with one of the two versions of UML specifications: UML 1.5 or UML 2.0.
- **Implementation project.** Project file extension: `.bdsproj.csproj` (Visual C# .NET), and `.vbproj` (Visual Basic .NET). You can create models for language-specific projects. Modeling that complies with UML 1.5 specification is supported for C# and DelphiVisual Basic .NET projects. Together modeling features are automatically activated for these projects.

The set of available project types depends on your license. Together Designer is required to work with design projects. Together Developer is required to work with implementation projects.

There are multiple project formats of the types mentioned above, supported by Together:

Supported project formats

Project format	Project type	Basic supported actions
Delphi formats	Implementation	Create, open, save, edit
Delphi for .NET	Implementation	Create, open, save, edit
C++Builder	Implementation	Create, open, save, edit
Together design formats: UML 1.5, UML 2.0	Design	Create, open, save, edit
Other editions of Together	Design or implementation	Import, share
IBM Rational Rose (MDL) format	Design	Create a new design project by using the import wizard
XMI format	Design	Import, export

See Also

Modeling Project Overview (↗ see page 89)

Transformation of Design Project to Source Code Overview (↗ see page 94)

Creating a Project (↗ see page 264)

3.5.6 UML 1.5 Reference

This section contains reference material about UML 1.5 diagrams.

Topics

Name	Description
UML 1.5 Activity Diagrams (↗ see page 1117)	This section describes the elements of UML 1.5 Activity Diagrams.
UML 1.5 Class Diagrams (↗ see page 1121)	This section describes the elements of UML 1.5 Class Diagrams.
UML 1.5 Component Diagrams (↗ see page 1128)	This section describes the elements of UML 1.5 Component Diagrams.
UML 1.5 Deployment Diagrams (↗ see page 1129)	This section describes the elements of UML 1.5 Deployment Diagrams.
UML 1.5 Interaction Diagrams (↗ see page 1131)	This section describes the elements of UML 1.5 Sequence and Collaboration diagrams.

UML 1.5 Statechart Diagrams (↗ see page 1135)	This section describes the elements of UML 1.5 Statechart diagrams.
UML 1.5 Use Case Diagrams (↗ see page 1137)	This section describes the elements of UML 1.5 Use Case Diagrams.

3.5.6.1 UML 1.5 Activity Diagrams

This section describes the elements of UML 1.5 Activity Diagrams.

Topics

Name	Description
Deferred Event (↗ see page 1117)	A **deferred event** is like an internal transition that handles the event and places it in an internal queue until it is used or discarded. A deferred event may be thought of as an internal transition that handles the event and places it in an internal queue until it is used or discarded. You can add a deferred event to a state or activity element.
State (↗ see page 1117)	A state models a situation during which some (usually implicit) invariant condition holds. The invariant may represent a static situation such as an object waiting for some external event to occur. However, it can also model dynamic conditions such as the process of performing some activity (for example, the model element under consideration enters the state when the activity commences and leaves it as soon as the activity is completed).
Transition (↗ see page 1118)	A single transition comes out of each state or activity, connecting it to the next state or activity. A transition takes operation from one state to another and represents the response to a particular event. You can connect states with transitions and create internal transitions within states.
UML 1.5 Activity Diagram Definition (↗ see page 1119)	This topic describes the UML 1.5 Activity Diagram.
UML 1.5 Activity Diagram Elements (↗ see page 1120)	The table below lists the elements of UML 1.5 Activity diagrams that are available using the Tool PaletteToolbox. ***UML 1.5 activity diagram elements***

3.5.6.1.1 Deferred Event

A **deferred event** is like an internal transition that handles the event and places it in an internal queue until it is used or discarded.

A deferred event may be thought of as an internal transition that handles the event and places it in an internal queue until it is used or discarded. You can add a deferred event to a state or activity element.

See Also

UML 1.5 Statechart diagram (↗ see page 1135)

UML 1.5 Activity diagram (↗ see page 1117)

UML 2.0 State Machine diagram (↗ see page 1155)

3.5.6.1.2 State

A state models a situation during which some (usually implicit) invariant condition holds. The invariant may represent a static situation such as an object waiting for some external event to occur. However, it can also model dynamic conditions such as the process of performing some activity (for example, the model element under consideration enters the state when the activity commences and leaves it as soon as the activity is completed).

Actions

Entry and exit actions are executed when entering or leaving a state, respectively.

You can create these actions in statechart diagrams as special nodes, or as stereotyped internal transitions.

Composite (nested) state

Create a composite state by nesting one or more levels of states within one state. You can also place start/end states and a history state inside of a state, and draw transitions among the contained substates.

See Also

UML 1.5 Statechart Diagram Reference (see page 1135)

UML 1.5 Activity Diagram Reference (see page 1117)

UML 2.0 State Machine Diagram Reference (see page 1155)

3.5.6.1.3 Transition

A single transition comes out of each state or activity, connecting it to the next state or activity.

A transition takes operation from one state to another and represents the response to a particular event. You can connect states with transitions and create internal transitions within states.

Internal transition

An internal transition is a way to handle events without leaving a state (or activity) and dispatching its exit or entry actions. You can add an internal transition to a state or activity element.

An internal transition is shorthand for handling events without leaving a state and dispatching its exit or entry actions.

Self-transition

A self-transition flow leaves the state (or activity) dispatching any exit action(s), then reenters the state dispatching any entry action(s). You can draw self-transitions for both activity and state elements on an Activity Diagram.

Self-transition for Statechart Diagrams

Self-transition for Activity Diagrams

Multiple transition

A transition can branch into two or more mutually-exclusive transitions.

A transition may fork into two or more parallel activities. A solid bar indicates a fork and the subsequent join of the threads coming out of the fork.

A transition may have multiple sources (a join from several concurrent states) or it may have multiple targets (a fork to several concurrent states).

You can show multiple transitions with either a vertical or horizontal orientation in your State and Activity Diagrams. Both the State and Activity Diagram toolbars provide separate horizontal and vertical fork/join buttons for each orientation. The two orientations are semantically identical.

Guard expressions

All transitions, including internal ones, are provided with the guard conditions (logical expressions) that define whether this transition should be performed. Also you can associate a transition with an effect, which is an optional activity performed when the transition fires. The guard condition is enclosed in the brackets (for example, "`[false]`") and displayed near the transition link on a diagram. Effect activity is displayed next to the guard condition. You can define the guard condition and effect using the Object InspectorProperties Window.

Guard expressions (inside `[ ]`) label the transitions coming out of a branch. The hollow diamond indicates a branch and its subsequent merge that indicates the end of the branch.

See Also

UML 1.5 Statechart Diagram (see page 1135)

UML 1.5 Activity Diagram (see page 1117)

UML 2.0 State Machine Diagram (see page 1155)

3.5.6.1.4 UML 1.5 Activity Diagram Definition

This topic describes the UML 1.5 Activity Diagram.

Definition

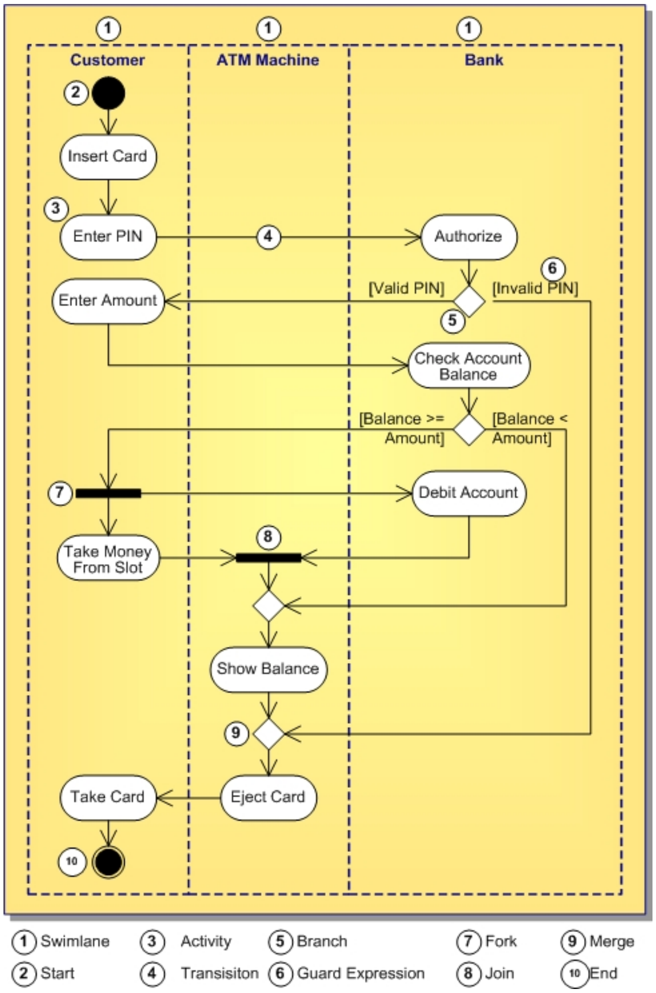
An Activity diagram is similar to a flowchart. Activity diagrams and Statechart diagrams are related. While a Statechart diagram focuses attention on an object undergoing a process (or on a process as an object), an Activity diagram focuses on the flow of activities involved in a single process. The Activity diagram shows how these single-process activities depend on one another.

Activity diagrams can be divided into object swimlanes that determine which object is responsible for an activity.

Sample Diagram

The Activity Diagram below uses the following process: "Withdraw money from a bank account through an ATM." The three involved classes of the activity are Customer, ATM Machine, and Bank. The process begins at the black start circle at the top and ends at the concentric white/black stop circles at the bottom. The activities are shown as rounded rectangles.

The three involved classes (people, and so on) of the activity are Customer, ATM, and Bank. The process begins at the black start circle at the top and ends at the concentric white/black stop circles at the bottom. The activities are shown as rounded rectangles.



3.5.6.1.5 UML 1.5 Activity Diagram Elements

The table below lists the elements of UML 1.5 Activity diagrams that are available using the Tool PaletteToolbox.

UML 1.5 activity diagram elements

Name	Type
Activity	node
Decision	node
Signal Receipt	node
Signal Sending	node
State	node
Start State	node
End State	node
History	node
Object	node
Transition	link

Horizontal Fork/Join	node
Vertical Fork/Join	node
Swimlane	node
Object Flow	link
Node by Pattern	opens Pattern Wizard (↗ see page 1162)
Link by Pattern	opens Pattern Wizard (↗ see page 1162)
Note	annotation
Note Link	annotation link

3.5.6.2 UML 1.5 Class Diagrams

This section describes the elements of UML 1.5 Class Diagrams.

Topics

Name	Description
Association Class and N-ary Association (↗ see page 1122)	Association classes appear in diagrams as three related elements: • Association class itself (represented by a class icon) • N-ary association class link (represented by a diamond) • Association connector (represented by a link between both) Association classes can connect to as many association end classes (participants) as required. The Object InspectorProperties Window of an association class, association link, and connector contain an additional Association tab. This tab contains the only label property, its value being synchronized with the name of the association class. For the association classes and association end links, the Custom node of the Object InspectorProperties Window displays... more (↗ see page 1122)
Class Diagram Relationships (↗ see page 1123)	There are several kinds of relationships: • **Association**: A relationship between instances of the two classes. There is an association between two classes if an instance of one class must know about the other to perform its work. In a diagram, an association is a link connecting two classes. Associations can be directed or undirected. A directed link points to the supplier class (the target). An association has two ends. An end may have a role name to clarify the nature of the association. A navigation arrow on an association shows which direction the association can be traversed or queried.... more (↗ see page 1123)

Class Diagram Types (see page 1124)	There are two types of class diagrams used in Together: - **Package (namespace)** diagrams. These diagrams are referred to as package diagrams in design projects, and namespace diagrams in implementation projects. They are stored as XML files in the `ModelSupport_%PROJECTNAME%ModelSupport` folder of the project groupsolution with the file extension `.txvpck`. - **Logical** class diagrams. These diagrams are stored as XML files in the `ModelSupport_%PROJECTNAME%ModelSupport` folder of the project groupsolution with the file extension `.txvcls`. Together automatically creates a default namespace diagram for the project and for each subdirectory under the project directory. The default project diagram... more (see page 1124)
Inner Classifiers (see page 1124)	The table below lists the diagram container elements along with the inner classifiers that you can add to container elements. ***Inner classifiers***
LiveSource Rules (see page 1125)	The impact of changing a class, interface, or namespace on a logical class diagram varies according to the kind of change: - Changing the name, adding a member, creating a new link, or applying a pattern makes the corresponding change in the actual source code. - Choose Delete from View on the context menu of the element to remove the element from a current diagram and keep the element in the namespace (package). - Choose Delete on the context menu to completely remove the element from the model. - When you press `Delete` on the keyboard, the Delete from view command is applied, if... more (see page 1125)
Members (see page 1125)	Note that the set of available members is different for the design and implementation projects. The table below lists the diagram elements along with the members that can be added using the context menu of the element. The type of applicable project is specified in square brackets. ***Members available***
UML 1.5 Class Diagram Definition (see page 1126)	Using Together, you can create language-neutral class diagrams in design projects, or language-specific class diagrams in implementation projects. For implementation projects, all diagram elements are immediately synchronized with the source code.
UML 1.5 Class Diagram Elements (see page 1127)	The table below lists the elements of UML 1.5 class diagrams that are available using the Tool PaletteToolbox. ***UML 1.5 class diagram elements***

3.5.6.2.1 Association Class and N-ary Association

Association classes appear in diagrams as three related elements:

- Association class itself (represented by a class icon)
- N-ary association class link (represented by a diamond)
- Association connector (represented by a link between both)

Association classes can connect to as many association end classes (participants) as required.

The Object InspectorProperties Window of an association class, association link, and connector contain an additional Association tab. This tab contains the only label property, its value being synchronized with the name of the association class. For the association classes and association end links, the Custom node of the Object InspectorProperties Window displays additional properties that corresponds to the role of this part of n-ary association (`associationClass` and `associationEnd` respectively).

You can delete each of the association end links or participant classes without destroying the entire n-ary association. However, deleting the association class results in deleting all the components of the n-ary association.

See Also

Creating an Association Class (see page 239)

Class Diagram Relationships (see page 1123)

UML 2.0 Class Diagram (see page 1143)

UML 1.5 Class Diagram (see page 1121)

3.5.6.2.2 Class Diagram Relationships

There are several kinds of relationships:

- **Association**: A relationship between instances of the two classes. There is an association between two classes if an instance of one class must know about the other to perform its work. In a diagram, an association is a link connecting two classes. Associations can be directed or undirected. A directed link points to the supplier class (the target). An association has two ends. An end may have a role name to clarify the nature of the association. A navigation arrow on an association shows which direction the association can be traversed or queried. A class can be queried about its Item, but not the other way around. The arrow also lets you know who "owns" the implementation of the association. Associations with no navigation arrows are bi-directional.
- **Generalization/Implementation**: An inheritance link indicating that a class implements an interface. An implementation has a triangle pointing to the interface.
- **Dependency**

There are several subtypes of an association relationship:

- **Simple Association**
- **Aggregation**: An association in which one class belongs to a collection. An aggregation has a diamond end pointing to the part containing the whole.
- **Composition**

Every class diagram has classes and associations. Navigability, roles, and multiplicities are optional items placed in a diagram to provide clarity.

The multiplicity of an association end is the number of possible instances of the class associated with a single instance of the other end. Multiplicities are single numbers or ranges of numbers. This table lists the most common multiplicities:

Multiplicities

Multiplicity	Meaning
0..1	Zero or one instance. The notation n . . m indicates n to m instances
0..* or *	No limit on the number of instances (including none)
1	Exactly one instance
1..*	At least one instance

See Also

UML 1.5 Class diagrams (see page 1121)

UML 2.0 Class diagrams (see page 1143)

3.5.6.2.3 Class Diagram Types

There are two types of class diagrams used in Together:

- **Package (namespace)** diagrams. These diagrams are referred to as package diagrams in design projects, and namespace diagrams in implementation projects. They are stored as XML files in the `ModelSupport_%PROJECTNAME%ModelSupport` folder of the project groupsolution with the file extension `.txvpck`.
- **Logical** class diagrams. These diagrams are stored as XML files in the `ModelSupport_%PROJECTNAME%ModelSupport` folder of the project groupsolution with the file extension `.txvcls`.

Together automatically creates a default namespace diagram for the project and for each subdirectory under the project directory. The default project diagram is named default; the default namespace (package) diagrams are named after the respective namespaces (packages).

You create logical class diagrams manually by using the **Add ▶ Class Diagram** or **Add ▶ Other Diagram** command on the project context menu.

See Also

UML 1.5 Class Diagram (see page 1121)

UML 2.0 ?lass Diagram (see page 1143)

3.5.6.2.4 Inner Classifiers

The table below lists the diagram container elements along with the inner classifiers that you can add to container elements.

Inner classifiers

Container element	Inner classifiers available
Class	Class Interface Structure [C#, Visual Basic] Delegate [C#, Visual Basic] Delegate as Function [Visual Basic] Enum [C#, Visual Basic]
Interface	Class [Visual Basic] Interface [Visual Basic] Delegate [Visual Basic] Delegate as Function [Visual Basic] Enum [Visual Basic]
Structure	Class Interface Structure Delegate Delegate as Function [Visual Basic] Enum

Module [Visual Basic]	Class [Visual Basic] Interface [Visual Basic] Structure [Visual Basic] Delegate [Visual Basic] Delegate as Function [Visual Basic] Enum [Visual Basic]

See Also

UML 1.5 Class Diagram Reference (see page 1121)

UML 2.0 Class Diagram Reference (see page 1143)

3.5.6.2.5 LiveSource Rules

The impact of changing a class, interface, or namespace on a logical class diagram varies according to the kind of change:

- Changing the name, adding a member, creating a new link, or applying a pattern makes the corresponding change in the actual source code.
- Choose Delete from View on the context menu of the element to remove the element from a current diagram and keep the element in the namespace (package).
- Choose Delete on the context menu to completely remove the element from the model.
- When you press `Delete` on the keyboard, the Delete from view command is applied, if it is available in this particular situation. If it is not, the element is deleted completely.
- Direct changes in source code editor, such as renaming a class, cannot be tracked by Together. Use refactoring operations for this purpose.

See Also

LiveSource Overview (see page 93)

UML 1.5 Class Diagram Reference (see page 1121)

3.5.6.2.6 Members

Note that the set of available members is different for the design and implementation projects.

The table below lists the diagram elements along with the members that can be added using the context menu of the element. The type of applicable project is specified in square brackets.

Members available

Container element	Members available
Class	Function [Visual Basic] Subroutine [Visual Basic] Method [C#] Operation [Design] Constructor [Design, C#, Visual Basic] Destructor [C#] Field [C#, Visual Basic] Attribute [Design] Property [C#, Visual Basic] Indexer [C#] Event [C#]
Interface	Function [Visual Basic] Subroutine [Visual Basic] Method [C#] Property [C#, Visual Basic] Indexer [C#] Event [C#] Attribute [Design] Operation [Design]
Structure	Method Constructor Field Property Indexer Event
Module	Function Subroutine Constructor Field Property
Enumeration	Enum Value

For implementation projects: If you set the abstract property for a method, property, or indexer (in abstract classes) as True in the Properties Window, the method body is removed from the source code. This is the desired behavior. Resetting the abstract property to False in the Object InspectorProperties Window, adds a new empty method body.

See Also

Adding a Member to a Container (see page 241)

UML 1.5 Class Diagram (see page 1121)

UML 2.0 Class Diagram (see page 1143)

3.5.6.2.7 UML 1.5 Class Diagram Definition

Using Together, you can create language-neutral class diagrams in design projects, or language-specific class diagrams in implementation projects. For implementation projects, all diagram elements are immediately synchronized with the source code.

Definition

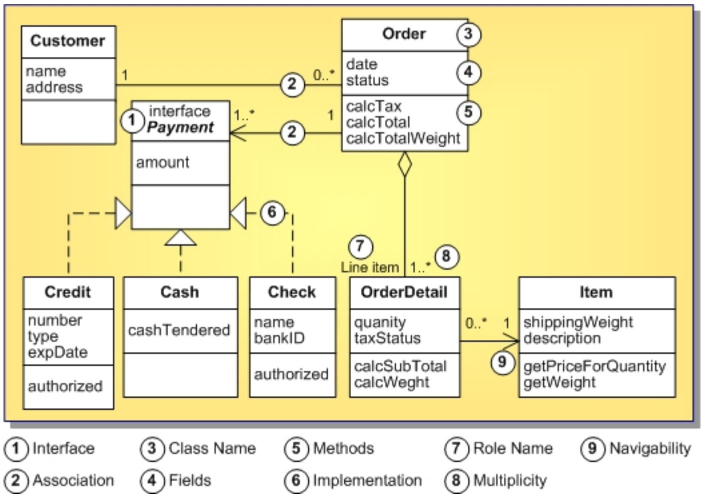
A class diagram provides an overview of a system by showing its classes and the relationships among them. Class diagrams are static: they display what interacts but not what happens during the interaction.

UML class notation is a rectangle divided into three parts: class name, fields, and methods. Names of abstract classes and interfaces are in italics. Relationships between classes are the connecting links.

In Together, the rectangle is further divided with separate partitions for properties and inner classes.

Sample Diagram

The following class diagram models a customer order from a retail catalog. The central class is the `Order`. Associated with it are the `Customer` making the purchase and the `Payment`. There are three types of payments: `Cash`, `Check`, or `Credit`. The order contains `OrderDetails` (line items), each with its associated Item.



There are three kinds of relationships used in this example:

- Association: For example, an `OrderDetail` is a line item of each `Order`.
- Aggregation: In this diagram, `Order` has a collection of `OrderDetails`.
- Implementation: `Payment` is an interface for `Cash`, `Check`, and `Credit`.

3.5.6.2.8 UML 1.5 Class Diagram Elements

The table below lists the elements of UML 1.5 class diagrams that are available using the Tool PaletteToolbox.

UML 1.5 class diagram elements

Name	Type
Namespace (Package)	node
Class	node
Interface	node
Association Class	node
Structure	node
Enumeration	node
Delegate	node

Object	node
Generalization/Implementation	link
Association	link
Dependency	link
Node by Pattern	opens Pattern Wizard (see page 1162)
Link by Pattern	opens Pattern Wizard (see page 1162)
Constraint	OCL node
Constraint link	OCL link
Note	annotation
Note Link	annotation link

3.5.6.3 UML 1.5 Component Diagrams

This section describes the elements of UML 1.5 Component Diagrams.

Topics

Name	Description
UML 1.5 Component Diagram Definition (see page 1128)	Both component and deployment diagrams depict the physical architecture of a computer-based system. Component diagrams show the dependencies and interactions between software components.
UML 1.5 Component Diagram Elements (see page 1129)	The table below lists the elements of UML 1.5 compoment diagrams that are available using the Tool PaletteToolbox. ***UML 1.5 component diagram elements***

3.5.6.3.1 UML 1.5 Component Diagram Definition

Both component and deployment diagrams depict the physical architecture of a computer-based system. Component diagrams show the dependencies and interactions between software components.

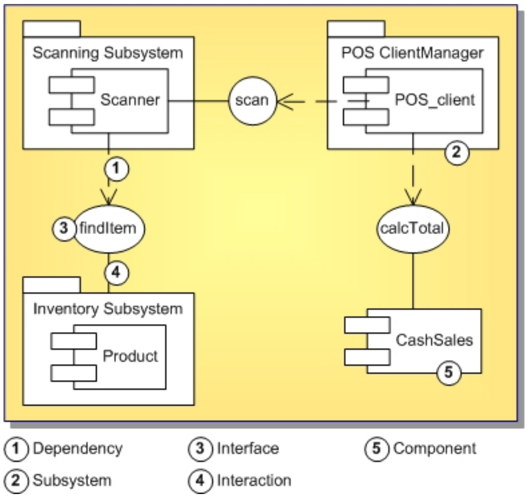
Definition

A component is a container of logical elements and represents things that participate in the execution of a system. Components also use the services of other components through one of its interfaces. Components are typically used to visualize logical packages of source code (work product components), binary code (deployment components), or executable files (execution components).

Sample Diagram

Following is a component diagram that shows the dependencies and interactions between software components for a cash register program.

3



3.5.6.3.2 UML 1.5 Component Diagram Elements

The table below lists the elements of UML 1.5 compoment diagrams that are available using the Tool PaletteToolbox.

UML 1.5 component diagram elements

Name	Type
Subsystem	node
Component	node
Interface	node
Supports	link
Dependency	link
Node by Pattern	opens Pattern Wizard (see page 1162)
Link by Pattern	opens Pattern Wizard (see page 1162)
Note	annotation
Note Link	annotation link

3.5.6.4 UML 1.5 Deployment Diagrams

This section describes the elements of UML 1.5 Deployment Diagrams.

Topics

Name	Description
UML 1.5 Deployment Diagram Definition (↗ see page 1130)	Both component and Deployment Diagrams depict the physical architecture of a computer-based system. Deployment Diagrams are made up of a graph of nodes connected by communication associations to show the physical configuration of the software and hardware. Components are physical units of packaging in software, including: • External libraries • Operating systems • Virtual machines
UML 1.5 Deployment Diagram Elements (↗ see page 1131)	The table below lists the elements of UML 1.5 deployment diagrams that are available using the Tool PaletteToolbox. ***UML 1.5 deployment diagram elements***

3.5.6.4.1 UML 1.5 Deployment Diagram Definition

Both component and Deployment Diagrams depict the physical architecture of a computer-based system.

Deployment Diagrams are made up of a graph of nodes connected by communication associations to show the physical configuration of the software and hardware.

Components are physical units of packaging in software, including:

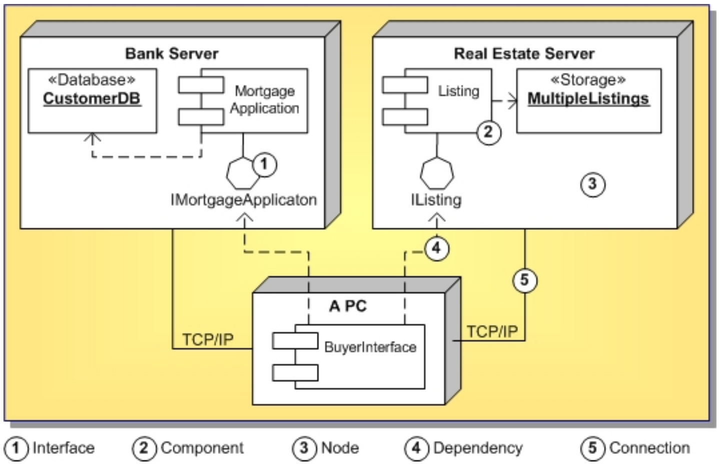
- External libraries
- Operating systems
- Virtual machines

Definition

The physical hardware is made up of nodes. Each component belongs on a node. Components are shown as rectangles with two tabs at the upper left.

Sample Diagram

Following is a Deployment Diagram that shows the relationships of software and hardware components for a real estate transaction.



3.5.6.4.2 UML 1.5 Deployment Diagram Elements

The table below lists the elements of UML 1.5 deployment diagrams that are available using the Tool PaletteToolbox.

UML 1.5 deployment diagram elements

Name	Type
Node	node
Component	node
Interface	node
Supports	link
Aggregates	link
Object	node
Association	link
Dependency	link
Node by Pattern	opens Pattern Wizard (see page 1162)
Link by Pattern	opens Pattern Wizard (see page 1162)
Note	annotation
Note Link	annotation link

3.5.6.5 UML 1.5 Interaction Diagrams

This section describes the elements of UML 1.5 Sequence and Collaboration diagrams.

Topics

Name	Description
Activation Bar (see page 1132)	Together automatically renders the activation of messages that show the period of time that the message is active. When you draw a message link to the destination object, the activation bar is created automatically. You can extend or reduce the period of time of a message by vertically dragging the top or bottom line of the activation bar as required. A longer activation bar means a longer time period when the message is active.

Conditional Block (↗ see page 1132)	Conditional block statement is a flexible tool to enhance a sequence diagram. The following statements are supported: • if • else • for • foreach • while • do while • try • catch • finally • switch • case • default
UML 1.5 Message (↗ see page 1133)	By default, message links in a sequence diagram are numbered sequentially from top to bottom. You can reorder messages. A “self message” is a message from an object back to itself.
Nested Message (↗ see page 1133)	You can nest messages by originating message links from an activation bar. The nested message inherits the numbering of the parent message. For example, if the parent message has the number 1, its first nested message is 1.1. It is also possible to create message links back to the parent activation bars.
UML 1.5 Collaboration Diagram Definition (↗ see page 1133)	Class diagrams are static model views. In contrast, interaction diagrams are dynamic, describing how objects collaborate.
UML 1.5 Interaction Diagram Elements (↗ see page 1134)	The table below lists the elements of UML 1.5 Interaction (Sequence and Collaboration) diagrams that are available using the Tool PaletteToolbox. ***UML 1.5 interaction diagram elements***
UML 1.5 Sequence Diagram Definition (↗ see page 1134)	Class diagrams are static model views. In contrast, interaction diagrams are dynamic, describing how objects collaborate.

3.5.6.5.1 Activation Bar

Together automatically renders the activation of messages that show the period of time that the message is active. When you draw a message link to the destination object, the activation bar is created automatically.

You can extend or reduce the period of time of a message by vertically dragging the top or bottom line of the activation bar as required. A longer activation bar means a longer time period when the message is active.

3.5.6.5.2 Conditional Block

Conditional block statement is a flexible tool to enhance a sequence diagram. The following statements are supported:

- if
- else
- for
- foreach
- while
- do while
- try
- catch

- finally
- switch
- case
- default

3.5.6.5.3 UML 1.5 Message

By default, message links in a sequence diagram are numbered sequentially from top to bottom. You can reorder messages.

A “self message” is a message from an object back to itself.

See Also

Messages in UML 2.0 (↗ see page 1153)

3.5.6.5.4 Nested Message

You can nest messages by originating message links from an activation bar. The nested message inherits the numbering of the parent message.

For example, if the parent message has the number 1, its first nested message is 1.1. It is also possible to create message links back to the parent activation bars.

See Also

UML 1.5 message (↗ see page 1133)

3.5.6.5.5 UML 1.5 Collaboration Diagram Definition

Class diagrams are static model views. In contrast, interaction diagrams are dynamic, describing how objects collaborate.

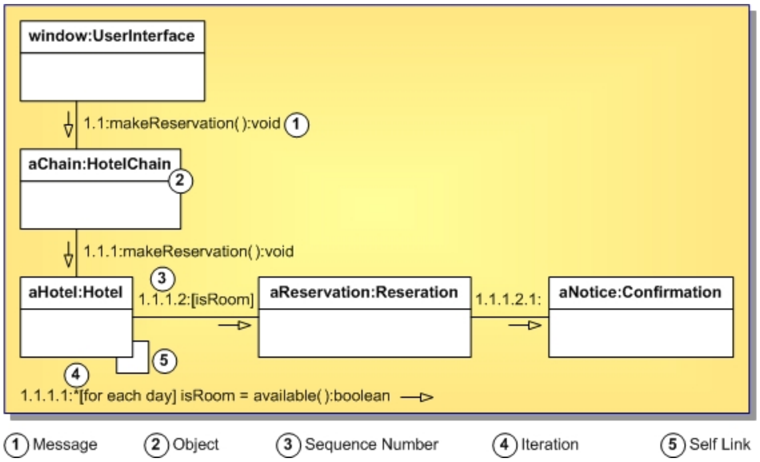
Definition

Like sequence diagrams, collaboration diagrams are also interaction diagrams. Collaboration diagrams convey the same information as sequence diagrams, but focus on object roles instead of the times that messages are sent.

In a sequence diagram, object roles are the vertices and messages are the connecting links. In a collaboration diagram, as follows, the object-role rectangles are labeled with either class or object names (or both). Colons precede the class names (`:`).

Sample Diagram

Each message in a collaboration diagram has a sequence number. The top-level message is numbered 1. Messages at the same level (sent during the same call) have the same decimal prefix but suffixes of 1, 2, etc. according to when they occur.



3.5.6.5.6 UML 1.5 Interaction Diagram Elements

The table below lists the elements of UML 1.5 Interaction (Sequence and Collaboration) diagrams that are available using the Tool PaletteToolbox.

UML 1.5 interaction diagram elements

Name	Type
Object	node
Actor	node
Message	link
Self Message	link
Message with delivery time	link, Sequence only
Conditional Block	node, Sequence only
Return	link, Sequence only
Association	link, Collaboration only
Aggregates	link, Collaboration only
Node by Pattern	opens Pattern Wizard (↗ see page 1162)
Link by Pattern	opens Pattern Wizard (↗ see page 1162)
Note	annotation
Note Link	annotation link

You can add shortcuts to the interaction diagrams, by using the **Add ▶ Shortcut** command. However, referring to the elements of the other interaction diagrams is not allowed.

3.5.6.5.7 UML 1.5 Sequence Diagram Definition

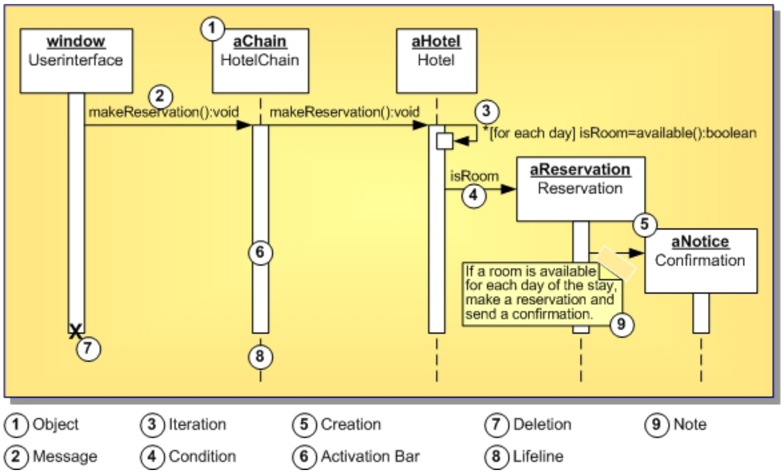
Class diagrams are static model views. In contrast, interaction diagrams are dynamic, describing how objects collaborate.

Definition

A sequence diagram is an interaction diagram that details how operations are carried out: what messages are sent and when. Sequence diagrams are organized according to time. The time progresses as you go down the page. The objects involved in the operation are listed from left to right according to when they take part in the message sequence.

Sample Diagram

Following is a Sequence Diagram for making a hotel reservation. The object initiating the sequence of messages is a Reservation window (the UserInterface).



The `UserInterface` sends a `makeReservation()` message to a `HotelChain`. The `HotelChain` then sends a `makeReservation()` message to a `Hotel`. If the Hotel has available rooms, then it makes a `Reservation` and a `Confirmation`.

Each vertical dotted line is a lifeline, representing the time that an object exists. Each arrow is a message call. An arrow goes from the sender to the top of the activation bar of the message on the receiver's lifeline. The activation bar represents the duration of execution of the message.

In this diagram, the `Hotel` issues a self call to determine if a room is available. If so, then the `Hotel` creates a `Reservation` and a `Confirmation`. The asterisk on the self call means iteration (to make sure there is available room for each day of the stay in the hotel). The expression in square brackets, `[ ]`, is a condition.

The diagram has a clarifying note, which is text inside a dog-eared rectangle. Notes can be included in any kind of UML diagram.

3.5.6.6 UML 1.5 Statechart Diagrams

This section describes the elements of UML 1.5 Statechart diagrams.

Topics

Name	Description
UML 1.5 Statechart Diagram Definition (↗ see page 1135)	This topic describes the UML 1.5 Statechart Diagram.
UML 1.5 Statechart Diagram Elements (↗ see page 1136)	The table below lists the elements of UML 1.5 Statechart diagrams that are available using the Tool PaletteToolbox. ***UML 1.5 Statechart diagram elements***

3.5.6.6.1 UML 1.5 Statechart Diagram Definition

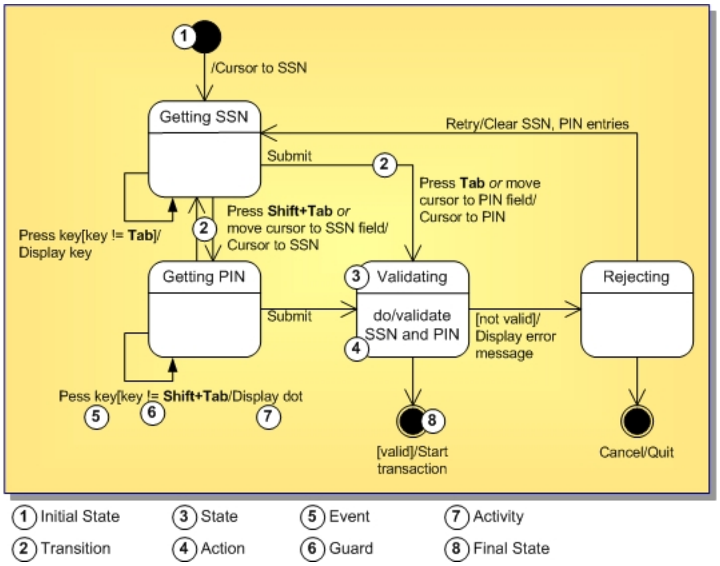
This topic describes the UML 1.5 Statechart Diagram.

Definition

Objects have behaviors and states. The state of an object depends on its current activity or condition. A statechart diagram shows the possible states of the object and the transitions that cause a change in state.

Sample Diagram

The following diagram models the login part of an online banking system. Logging in consists of entering a valid social security number and personal id number, then submitting the information for validation. Logging in can be factored into four non-overlapping states: `Getting SSN`, `Getting PIN`, `Validating`, and `Rejecting`. Each state provides a complete set of transitions that determines the subsequent state.


1
/Cursor to SSN
Getting SSN
Retry/Clear SSN, PIN entries
Submit
2
Press **Tab** *or* move cursor to PIN field/ Cursor to PIN
Press **Shift+Tab** *or* move cursor to SSN field/ Cursor to SSN
2
Press key[key != **Tab**]/ Display key
Getting PIN
3
Validating
Rejecting
Submit
do/validate SSN and PIN
4
[not valid]/ Display error message
Pess key[key != **Shift+Tab**/Display dot
5
6
7
8
[valid]/Start transaction
Cancel/Quit
1 Initial State
2 Transition
3 State
4 Action
5 Event
6 Guard
7 Activity
8 Final State

States are depicted as rounded rectangles. Transitions are arrows from one state to another. Events or conditions that trigger transitions are written next to the arrows. This diagram has two self-transitions: `Getting SSN` and `Getting PIN`. The initial state (shown as a black circle) is a dummy to start the action. Final states are also dummy states that terminate the action.

The action that occurs as a result of an event or condition is expressed in the form `/action`. While in its Validating state, the object does not wait for an outside event to trigger a transition. Instead, it performs an activity. The result of that activity determines its subsequent state.

3.5.6.6.2 UML 1.5 Statechart Diagram Elements

The table below lists the elements of UML 1.5 Statechart diagrams that are available using the Tool PaletteToolbox.

UML 1.5 Statechart diagram elements

Name	Type
State	node
Start State	node
End State	node
History	node
Object	node
Transition	link
Horizontal Fork/Join	node
Vertical Fork/Join	node
Node by Pattern	opens Pattern Wizard (↗ see page 1162)
Link by Pattern	opens Pattern Wizard (↗ see page 1162)

Note	annotation
Note Link	annotation link

3.5.6.7 UML 1.5 Use Case Diagrams

This section describes the elements of UML 1.5 Use Case Diagrams.

Topics

Name	Description
Extension Point (see page 1137)	An **extension point** refers to a location within a use case where you can insert action sequences from other use cases. An extension point consists of a unique name within a use case and a description of the location within the behavior of the use case. In a use case diagram, extension points are listed in the use case with the heading "Extension Points" (appears as bold text in the Diagram View).
UML 1.5 Use Case Diagram Definition (see page 1137)	Use case diagrams are helpful in three areas: • Determining features (requirements): New use cases often generate new requirements as the system is analyzed and the design takes shape. • Communicating with clients: Notational simplicity makes use case diagrams a good way for developers to communicate with clients. • Generating test cases: The collection of scenarios for a use case may suggest a suite of test cases for those scenarios.
UML 1.5 Use Case Diagram Elements (see page 1138)	The table below lists the elements of UML 1.5 Use Case diagrams that are available using the Tool PaletteToolbox. ***UML 1.5 Use Case diagram elements***

3.5.6.7.1 Extension Point

An **extension point** refers to a location within a use case where you can insert action sequences from other use cases.

An extension point consists of a unique name within a use case and a description of the location within the behavior of the use case.

In a use case diagram, extension points are listed in the use case with the heading "Extension Points" (appears as bold text in the Diagram View).

See Also

UML 1.5 Use case diagram (see page 1137)

UML 2.0 Use case diagram (see page 1157)

3.5.6.7.2 UML 1.5 Use Case Diagram Definition

Use case diagrams are helpful in three areas:

- Determining features (requirements): New use cases often generate new requirements as the system is analyzed and the design takes shape.
- Communicating with clients: Notational simplicity makes use case diagrams a good way for developers to communicate with clients.

- Generating test cases: The collection of scenarios for a use case may suggest a suite of test cases for those scenarios.

Definition

Use Case Diagrams describe what a system does from the viewpoint of an external observer. The emphasis is on what a system does rather than how.

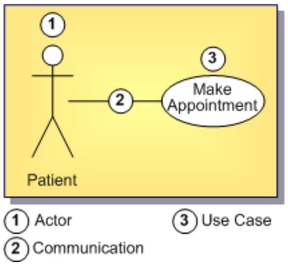
Use Case Diagrams are closely connected to scenarios. A scenario is an example of what happens when someone interacts with the system.

Sample Diagram

Following is a scenario for a medical clinic:

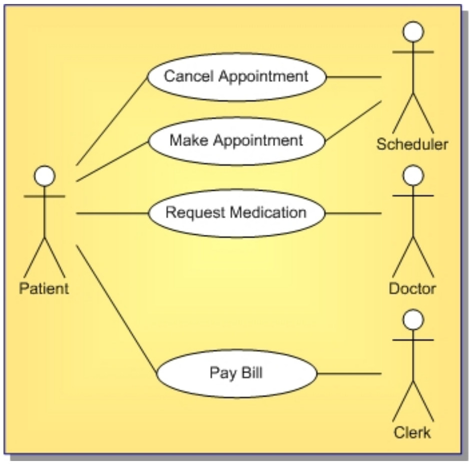
A patient calls the clinic to make an appointment for a yearly checkup. The receptionist finds the nearest empty time slot in the appointment book and schedules the appointment for that time slot.

A use case is a summary of scenarios for a single task or goal. An actor is who or what initiates the events involved in that task. Actors are simply roles that people or objects play. The following diagram is the `Make Appointment` use case for the medical clinic. The actor is a `Patient`. The connection between actor and use case is a communication association (or communication for short).



Actors are stick figures. Use cases are ovals. Communications are lines that link actors to use cases.

A use case diagram is a collection of actors, use cases, and their communications. Following is an example of the use case Make Appointment as part of a diagram with four actors and four use cases. Notice that a single use case can have multiple actors.



3.5.6.7.3 UML 1.5 Use Case Diagram Elements

The table below lists the elements of UML 1.5 Use Case diagrams that are available using the Tool PaletteToolbox.

UML 1.5 Use Case diagram elements

Name	Type
Actor	node
Use Case	node
Communicates	link
Extend	link
Include	link
Generalization	link
System Boundary	node
Node by Pattern	opens Pattern Wizard (↗ see page 1162)
Link by Pattern	opens Pattern Wizard (↗ see page 1162)
Note	annotation
Note Link	annotation link

3.5.7 Together Glossary

This topic contains a dictionary of specific terms used in Together user interface and documentation. This dictionary is sorted alphabetically.

Term	Description
Cardinality	The number of elements in a set. See also **multiplicity**.
Classifier	In general, a classifier is a classification of instances — it describes a set of instances that have features in common. In Together, classifiers are the basic nodes of Class diagrams: class, interface, structure, delegate, enum, module. Some of them can include other classifiers, or **inner classifiers** (see Inner Classifiers (↗ see page 1124)).
Compartment	Some of Together model elements (basically, classes) are represented by rectangles with several **compartments** inside. You can change appearance of the compartments (see Diagram Appearance options (↗ see page 1089)).
Design project	One of the two basic project types supported by Together: **design** and **implementation**. A design project is language-neutral. It does not contain source code.
Diagram	A graphical presentation of a collection of model elements, most often rendered as a connected graph of arcs (relationships) and vertices (other model elements). The set of available diagram for a project depend on the project type.
Implementation project	One of the two basic project types supported by Together: **design** and **implementation**. An implementation project is language-specific. It includes diagrams and source code.
Invocation specification	Invocation specification is an area within an execution specification on a UML 2.0 Sequence Diagram. This element is not defined in the UML 2.0 specification, but introduced in Together. It is a useful tool for modeling synchronous invocations with the reply messages. A message in UML 2.0 Sequence Diagrams has its origin in an invocation specification.
Model element	Model element is any component of your model that you can put on a diagram. Model elements include **nodes** and **links** between them.

Multiplicity	A specification of the range of allowable cardinalities that a set may assume. Multiplicity specifications may be given for association ends, parts within composites, repetitions, and other purposes. A multiplicity is a subset of the non-negative integers. See also **cardinality**.
N-ary association	An association among three or more classes. Each instance of the association is an n-tuple of values from the respective classes.
Shortcut	A **shortcut** is a representation of an existing node element placed on the same or a different diagram.
View filter	A view filter is mechanism to show or hide a specific kind of model elements. When dealing with large projects, the amount of information shown on a diagram can become overwhelming. In Together, you can selectively show or hide information. See Using view filters (see page 233).

See Also

Help on Help (see page 51)

About Together (see page 83)

3.5.8 UML 2.0 Reference

This section contains reference material about UML 2.0 diagrams.

Topics

Name	Description
UML 2.0 Activity Diagrams (see page 1140)	This section describes the elements of UML 2.0 Activity Diagrams.
UML 2.0 Class Diagrams (see page 1143)	This section describes the elements of UML 2.0 Class diagrams.
UML 2.0 Component Diagrams (see page 1145)	This section describes the elements of UML 2.0 Component diagrams.
UML 2.0 Composite Structure Diagrams (see page 1146)	This section describes the elements of UML 2.0 Composite Structure Diagrams.
UML 2.0 Deployment Diagrams (see page 1148)	This section describes the elements of UML 2.0 Deployment diagrams.
UML 2.0 Interaction Diagrams (see page 1149)	This section describes the elements of UML 2.0 Communication and Sequence diagrams.
UML 2.0 State Machine Diagrams (see page 1155)	This section describes the elements of UML 2.0 State Machine Diagrams.
UML 2.0 Use Case Diagrams (see page 1157)	This section describes the elements of UML 2.0 Use Case Diagrams.

3.5.8.1 UML 2.0 Activity Diagrams

This section describes the elements of UML 2.0 Activity Diagrams.

Topics

Name	Description
Pin (see page 1141)	**Actions** can consume some input values or produce some output values. Input, Output and Value **pins** hold such values.
UML 2.0 Activity Diagram Definition (see page 1141)	
UML 2.0 Activity Diagram Elements (see page 1142)	The table below lists the elements of UML 2.0 Activity diagrams that are available using the Tool PaletteToolbox. ***UML 2.0 Activity diagram elements***

3.5.8.1.1 Pin

Actions can consume some input values or produce some output values. Input, Output and Value **pins** hold such values.

See Also

Creating Input Pins (see page 217)

UML 2.0 Activity Diagram Reference (see page 1140)

3.5.8.1.2 UML 2.0 Activity Diagram Definition

Definition

The activity diagram enables you to model the system behavior, including the sequence and conditions of execution of the actions. Actions are the basic units of the system behavior.

An Activity diagram enables you to group and ungroup actions. If an action can be broken into a sequence of other actions, you can create an activity to represent them.

In UML 2.0, activities consist of actions. Actions are not states (compared to UML 1.x) and can have subactions. An action represents a single step within an activity, that is, one that is not further decomposed within the activity. An activity represents a behavior which is composed of individual elements that are actions. An action is an executable activity node that is the fundamental unit of executable functionality in an activity, as opposed to control and data flow among actions. The execution of an action represents some transformation or processing in the modeled system, be it a computer system or otherwise.

The semantics of activities is based on token flow. By flow, we mean that the execution of one node affects and is affected by the execution of other nodes, and such dependencies are represented by edges in the activity diagram. Data and control flows are different in UML 2.0.

A control flow may have multiple sources (it joins several concurrent actions) or it may have multiple targets (it forks into several concurrent actions).

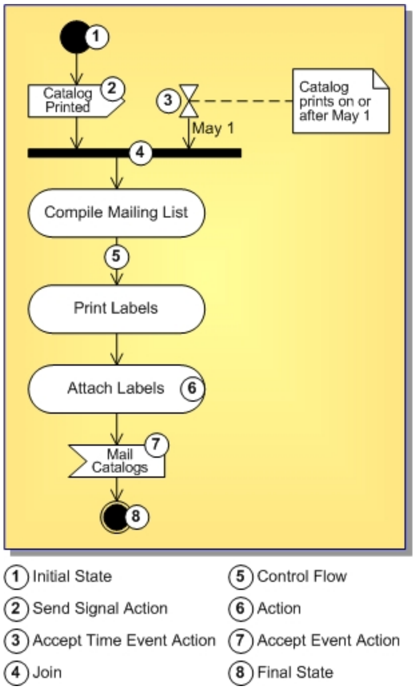
Each flow within an activity can have its own termination, which is denoted by a flow final node. The flow final node means that a certain flow within an activity is complete. Note that the flow final may not have any outgoing links.

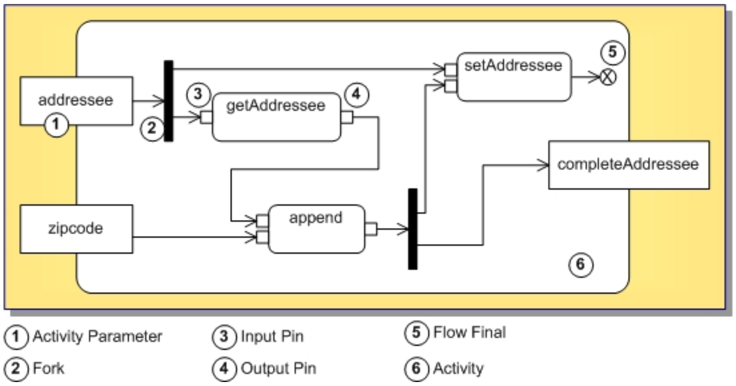
Using decisions and merges, you can manage multiple outgoing and incoming control flows.

Use the Object InspectorProperties Window to adjust action properties, including:

- In the Properties, View, Description, and Custom tabs, configure standard properties of the element.
- In the Local Precondition and Local Postcondition tabs, select the language of the constraint expression from the Language list box. The possible options are OCL and plain text. In the edit field below the list box, enter the constraint expression for this action.

Sample Diagram





3.5.8.1.3 UML 2.0 Activity Diagram Elements

The table below lists the elements of UML 2.0 Activity diagrams that are available using the Tool PaletteToolbox.

UML 2.0 Activity diagram elements

Name	Type
Activity	node
Activity parameter	node component
Action	node
Initial	node
Activity final	node

Decision	node
Merge	node
Flow Final	node
Control Flow	link
Input pin	pin
Output pin	pin
Value pin	pin
Object node	node
Central Buffer	node
Data Store	node
Object Flow	link
Accept Event Action	node
Accept Time Event Action	node
Send Signal Action	node
Node by Pattern	opens Pattern Wizard (see page 1162)
Link by Pattern	opens Pattern Wizard (see page 1162)
Note	annotation
Note Link	annotation link

3.5.8.2 UML 2.0 Class Diagrams

This section describes the elements of UML 2.0 Class diagrams.

Topics

Name	Description
UML 2.0 Class Diagram Definition (see page 1143)	UML 2.0 Class diagrams feature the same capabilities as the UML 1.5 diagrams. The UML 2.0 class diagrams offer new diagram elements such as ports, provided and required interfaces, and slots. According to the UML 2.0 specification, an instance specification can instantiate one or more classifiers. You can use classes, interfaces, or components as a classifier.
UML 2.0 Class Diagram Elements (see page 1144)	The table below lists the elements of UML 2.0 class diagrams that are available using the Tool PaletteToolbox. ***UML 2.0 class diagram elements***

3.5.8.2.1 UML 2.0 Class Diagram Definition

UML 2.0 Class diagrams feature the same capabilities as the UML 1.5 diagrams.

The UML 2.0 class diagrams offer new diagram elements such as ports, provided and required interfaces, and slots.

According to the UML 2.0 specification, an instance specification can instantiate one or more classifiers. You can use classes, interfaces, or components as a classifier.

Interfaces

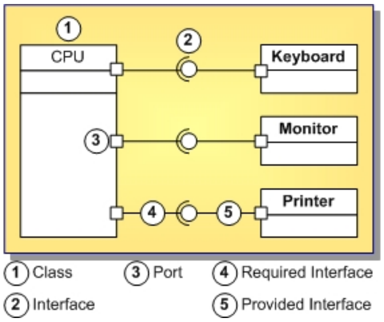
A class implements an interface via the same generalization/implementation link, as in UML 1.5 class diagram. In addition to the implementation interfaces, there are provided and required interfaces. Interfaces can be represented in class diagrams as rectangles or as circles. For the sake of clarity of your diagrams, you can show or conceal interfaces.

Tip: Applying a provided interface link between a class and an interface creates a regular generalization/implementation link. To create provided interface, apply the provided interface link to a port on the client class.

UML 2.0 class diagram supports the ball-and socket notation for the provided and required interfaces. Choose Show as circle command on the context menu of the interface to obtain a lollipop between the client class and the supplier interface.

Sample Diagram

The figure below shows a class diagram with some of the new elements.



3.5.8.2.2 UML 2.0 Class Diagram Elements

The table below lists the elements of UML 2.0 class diagrams that are available using the Tool PaletteToolbox.

UML 2.0 class diagram elements

Name	Type
Package	node
Class	node
Interface	node
Association Class	node
Port	node
Instance specification	node
Generalization/Implementation	link
Provided interface	link
Required interface	link
Association	link
Association end	link
Dependency	link
Node by Pattern	opens Pattern Wizard (↗ see page 1162)
Link by Pattern	opens Pattern Wizard (↗ see page 1162)
Constraint	OCL node
Constraint link	OCL link
Note	annotation
Note Link	annotation link

3.5.8.3 UML 2.0 Component Diagrams

This section describes the elements of UML 2.0 Component diagrams.

Topics

Name	Description
UML 2.0 Component Diagram Definition (↗ see page 1145)	This topic describes the UML 2.0 Component Diagram.
UML 2.0 Component Diagram Elements (↗ see page 1145)	The table below lists the elements of UML 2.0 compoment diagrams that are available using the Tool PaletteToolbox. ***UML 2.0 component diagram elements***

3.5.8.3.1 UML 2.0 Component Diagram Definition

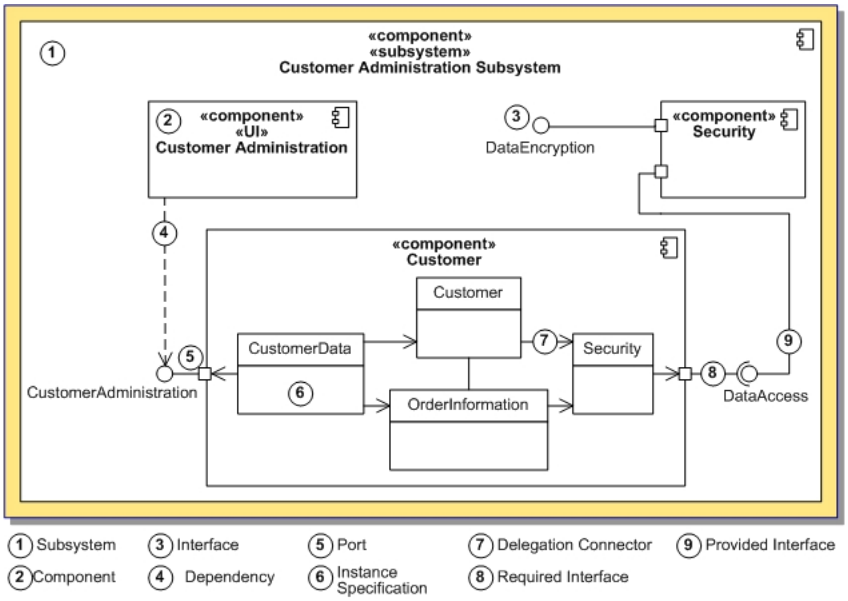
This topic describes the UML 2.0 Component Diagram.

Definition

According to the UML 2.0 specification, a component diagram can contain instance specifications. An instance specification can be defined by one or more classifiers. You can use classes, interfaces, or components as a classifiers. You can instantiate a classifier using the Object InspectorProperties Window, or the in-place editor.

Sample Diagram

The following component diagram specifies a set of constructs that can be used to define software systems of arbitrary size and complexity.



3.5.8.3.2 UML 2.0 Component Diagram Elements

The table below lists the elements of UML 2.0 compoment diagrams that are available using the Tool PaletteToolbox.

UML 2.0 component diagram elements

Name	Type
Component	node

Class	node
Port	node
Artifact	node
Interface	node
Instance specification	node
Delegation connector	link
Provided interface	link
Required interface	link
Association	link
Aggregation	link
Dependency	link
Realization	link
Node by Pattern	opens Pattern Wizard (see page 1162)
Link by Pattern	opens Pattern Wizard (see page 1162)
Note	annotation
Note Link	annotation link

3.5.8.4 UML 2.0 Composite Structure Diagrams

This section describes the elements of UML 2.0 Composite Structure Diagrams.

Topics

Name	Description
Delegation Connector (see page 1146)	An interface can delegate its obligations to another interface through the delegation connector.
UML 2.0 Composite Structure Diagram Definition (see page 1146)	Diagram courtesy of the Unified Modeling Language: *Superstructure version 2.0. August 2003. p. 178*.
UML 2.0 Composite Structure Diagram Elements (see page 1147)	The table below lists the elements of UML 2.0 composite structure diagrams that are available using the Tool PaletteToolbox. ***UML 2.0 composite structure diagram elements***

3.5.8.4.1 Delegation Connector

An interface can delegate its obligations to another interface through the delegation connector.

See Also

UML 2.0 Composite Structure Diagram Reference (see page 1146)

3.5.8.4.2 UML 2.0 Composite Structure Diagram Definition

Diagram courtesy of the Unified Modeling Language: *Superstructure version 2.0. August 2003. p. 178*.

Definition

Composite structure diagrams depict the internal structure of a classifier, including its interaction points to the other parts of the system. It shows the configuration of parts that jointly perform the behavior of the containing classifier.

A collaboration describes a structure of collaborating parts (roles). A collaboration is attached to an operation or a classifier through a Collaboration Use.

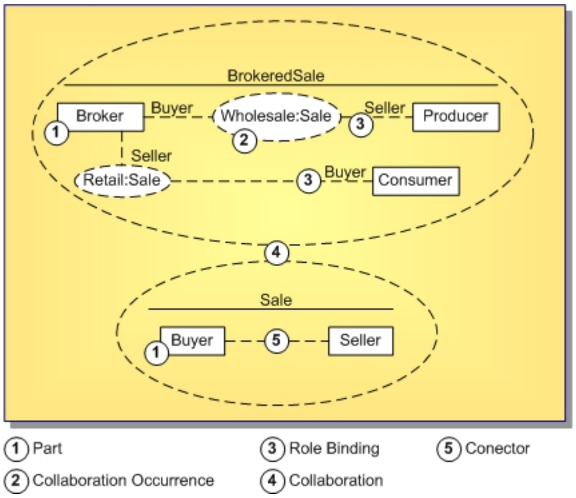
Classes and collaborations in the Composite Structure diagram can have internal structure and ports. Internal structure is represented by a set of interconnected parts (roles) within the containing class or collaboration. Participants of a collaboration or a class are linked by the connectors.

A port can appear either on a contained part, or on the boundary of the class.

The contained parts can be included by reference. Referenced parts are represented by the dotted rectangles.

Composite Structure diagram supports the ball-and-socket notation for the provided and required interfaces. Interfaces can be shown or hidden in the diagram as needed.

Sample Diagram



3.5.8.4.3 UML 2.0 Composite Structure Diagram Elements

The table below lists the elements of UML 2.0 composite structure diagrams that are available using the Tool PaletteToolbox.

UML 2.0 composite structure diagram elements

Name	Type
Class	node
Interface	node
Collaboration	node
Collaboration Occurrence	node
Part	node
Referenced part	node
Port	node
Provided interface	link
Required interface	link
Connector	link
Collaboration role	link
Role binding	link

Node by Pattern	opens Pattern Wizard (see page 1162)
Link by Pattern	opens Pattern Wizard (see page 1162)
Note	annotation
Note Link	annotation link

3.5.8.5 UML 2.0 Deployment Diagrams

This section describes the elements of UML 2.0 Deployment diagrams.

Topics

Name	Description
UML 2.0 Deployment Diagram Definition (see page 1148)	This topic describes the UML 2.0 Deployment Diagram.
UML 2.0 Deployment Diagram Elements (see page 1148)	The table below lists the elements of UML 2.0 deployment diagrams that are available using the Tool PaletteToolbox. ***UML 2.0 deployment diagram elements***

3.5.8.5.1 UML 2.0 Deployment Diagram Definition

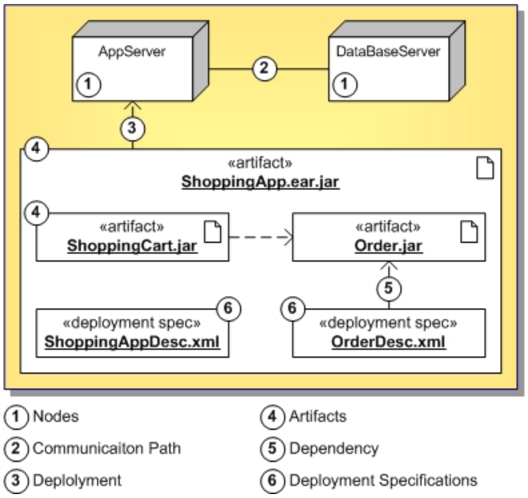
This topic describes the UML 2.0 Deployment Diagram.

Definition

The deployment diagram specifies a set of constructs that can be used to define the execution architecture of systems that represent the assignment of software artifacts to nodes. Nodes are connected through communication paths to create network systems of arbitrary complexity. Nodes are typically defined in a nested manner, and represent either hardware devices or software execution environments. Artifacts represent concrete elements in the physical world that are the result of a development process.

Diagram courtesy of the Unified Modeling Language: *Superstructure version 2.0. August 2003. pp. 207, 212*.

Sample Diagram



3.5.8.5.2 UML 2.0 Deployment Diagram Elements

The table below lists the elements of UML 2.0 deployment diagrams that are available using the Tool PaletteToolbox.

UML 2.0 deployment diagram elements

Name	Type
Node	node
Artifact	node
Device	node
Execution specification	node
Deployment specification	node
Instance specification	node
Deployment	link
Generalization	link
Association	link
Dependency	link
Manifestation	link
Communication path	link
Node by Pattern	opens Pattern Wizard (↗ see page 1162)
Link by Pattern	opens Pattern Wizard (↗ see page 1162)
Note	annotation
Note Link	annotation link

An artifact represents a physical entity and is depicted in diagram as a rectangle with the `<<artifact>>` stereotype. An artifact may have properties which define its features, and operations which can be performed on its instances. Physically the artifacts can be model files, source files, scripts, binary executable files, a table in a database system, a development deliverable, a word-processing document, or a mail message. A deployed artifact is one that has been deployed to a node used as a deployment target. Deployed artifacts are connected with the target node by deployment links.

Artifacts can include operations.

You can create complex artifacts, by nesting artifact icons.

3.5.8.6 UML 2.0 Interaction Diagrams

This section describes the elements of UML 2.0 Communication and Sequence diagrams.

Topics

Name	Description
Interaction (↗ see page 1150)	By using Together, you can create interactions for the detailed description and analysis of inter-process communications. Interactions can be visually represented in your Together projects by means of the two most common interaction diagrams: Sequence and Communication. On the other hand, interactions can exist in projects without visual representation.
Execution Specification and Invocation Specification (↗ see page 1151)	In sequence diagrams, Together automatically renders the execution specification of a message that shows the period of time when the message is active. When you draw a message link to the destination lifeline, the execution specification bar is created automatically. You can extend or reduce the period of time of a message by vertically dragging the top or bottom line of the execution specification as required. It is also possible to create an execution specification on a lifeline without creating an incoming message link. In this case a found message is created, that is a message that comes from an... more (↗ see page 1151)
Operator and Operand for a Combined Fragment (↗ see page 1151)	

UML 2.0 Communication Diagram Definition (↗ see page 1152)	Interactions can be visually represented in your Together projects by means of the two most common interaction diagrams: Sequence and Communication. Communication Diagrams focus on the interaction between lifelines where the architecture of the internal structure and how this corresponds with the message passing is central. The sequencing of messages is given through a sequence numbering scheme.
UML 2.0 Communication Diagram Elements (↗ see page 1152)	The table below lists the elements of UML 2.0 communication diagrams that are available using the Tool PaletteToolbox. ***UML 2.0 communication diagram elements***
UML 2.0 Message (↗ see page 1153)	Call messages are always visible in diagrams; reply messages normally are not displayed. However, you can visualize the reply message.
UML 2.0 Sequence Diagram Definition (↗ see page 1154)	Interactions can be visually represented in your Together projects by means of the two most common interaction diagrams: Sequence and Communication. The most common kind of Interaction Diagram is the Sequence Diagram, which focuses on the message interchange between a number of lifelines. A Sequence Diagram describes an interaction by focusing on the sequence of messages that are exchanged.
UML 2.0 Sequence Diagram Elements (↗ see page 1154)	The table below lists the elements of UML 2.0 sequence diagrams that are available using the Tool PaletteToolbox. ***UML 2.0 sequence diagram elements***

3.5.8.6.1 Interaction

By using Together, you can create interactions for the detailed description and analysis of inter-process communications.

Interactions can be visually represented in your Together projects by means of the two most common interaction diagrams: Sequence and Communication. On the other hand, interactions can exist in projects without visual representation.

Interaction use

Within an interaction, you can refer to the other interactions described in your project. So called “Interaction use” elements serve this purpose. Note that referenced interaction can be explicitly defined from the model, or just specified as a text string.

Each interaction use is attached to its lifeline with a black dot. This dot is an individual diagram element. If an interaction use is expanded over several lifelines, you can delete the attachment dots from all lifelines but one. An interaction use should be connected with at least one lifeline.

Tie frame

Together makes it possible to spread combined fragments and interaction uses across several lifelines. This is done with the Tie Frame button of the diagram Tool PaletteToolbox.

A frame can be attached to different points on the target lifeline. You choose the desired location on the target lifeline between the other elements and frames that belong to it. The frame shifts accordingly along the source lifeline.

It is important to understand that only those lifelines marked with dots are attached to the referenced interaction or combined fragment; lifelines that are only crossed by the frame are not attached. Attachment dots are separate diagram elements that can be selected and deleted.

Lifeline

A lifeline defines an individual participant of the interaction. A lifeline is shown in a sequence diagram as a rectangle followed by a vertical-dashed line.

Lifelines of an interaction can represent the parts defined in the class or composite structure diagrams. If the referenced element is multivalued, then the lifeline should have a selector that specifies which particular part is represented by this lifeline.

If a lifeline represents a connectable element, has type specified, or refers to another interaction, the Select menu becomes enabled on the context menu of this lifeline. Using this menu, you can navigate to the part, type or decomposition associated with the lifeline. These properties are defined by using the Object InspectorProperties Window. If the represents property is set, the type and part properties are disabled.

You can define these properties manually by typing the values in the corresponding fields of the Object InspectorProperties

Window. If the specified values are not found in the model, they are displayed in single quotes. Such references are not related to any actual elements and the Select menu is not available for them. If the specified values can be resolved in the model, they are shown without quotes, and the Select menu is available for them.

State invariant

A state invariant is a constraint placed on a lifeline. This constraint is evaluated at runtime prior to execution of the next execution specification. State invariants are represented in the interaction diagrams in two ways: as OCL expressions or as references to the state diagrams. You can use the state invariants to provide comments to your interaction diagrams and to connect interactions with states.

It is important to note that Together provides validation of the state invariants represented as OCL expressions. If the syntax is wrong, or there is no valid context, the constraint is displayed red. For example, a lifeline should have type and represents properties defined to be a valid context.

See Also

OCL support overview (see page 95)

UML 2.0 interaction diagrams (see page 1149)

3.5.8.6.2 Execution Specification and Invocation Specification

In sequence diagrams, Together automatically renders the execution specification of a message that shows the period of time when the message is active. When you draw a message link to the destination lifeline, the execution specification bar is created automatically. You can extend or reduce the period of time of a message by vertically dragging the top or bottom line of the execution specification as required.

It is also possible to create an execution specification on a lifeline without creating an incoming message link. In this case a found message is created, that is a message that comes from an object that is not shown in diagram. Use the Object InspectorProperties Window to hide or show the found messages.

Messages in sequence diagrams have their origin in an invocation specification. This is an area within an execution specification. The notion of an invocation specification is introduced in Together's implementation of UML 2.0 sequence diagrams. Though this element is not defined in the UML 2.0 specification, it is a useful tool for modeling synchronous invocations with the reply messages. In particular, invocation specification marks a place where the reply messages (even if they are invisible) enter the execution context of a lifeline, and where sub-messages may reenter the lifeline.

Active and passive areas of the execution specification are rendered in different colors. The white execution specification bars denote active areas where you can create message links. The gray bars are passive and are not a valid source or target for the message links.

See Also

UML 2.0 Message (see page 1153)

3.5.8.6.3 Operator and Operand for a Combined Fragment

About combined fragment

A combined fragment can consist of one or more interaction operators and one or more interaction operands. Number of interaction operands (just one, or more than one) depends on the last interaction operator of this combined fragment.

Use the Tool PaletteToolbox, or context menus to create these elements. The operator type shows up in the descriptor in the upper-left corner of the design element. Note that you can define multiple operators in a combined fragment. In this case, the descriptor contains the list of all operators, which is a shorthand for nested operators.

When an operator is created, add the allowed operands, using the combined fragment's context menu.

A combined fragment can be expanded over several lifelines, detached from and reattached to lifelines. In the Object InspectorProperties Window, use the Operators field to manage operators within the combined fragment.

Each combined fragment is attached to its lifeline with a black dot. This dot is an individual diagram element, which can be selected or deleted. Deleting a dot means detaching a combined fragment from the lifeline. Note that a combined fragment cannot be detached from all lifelines and should have at least one attachment dot.

You can reattach a combined fragment later, using the Tie Frame tool.

Operator

When a combined fragment is created, the operator is shown in a descriptor pentagon in the upper left corner of the frame. You can change the operator type, using the Operators field of the Object InspectorProperties Window, which is immediately reflected in the descriptor.

The descriptor may contain several operators. UML 2.0 specification provides this notation for the nested combined fragments. In Together you can use this notation, or create nested combined fragment nodes.

Operand

Operands are represented as rectangular areas within a combined fragment, separated by the dashed lines. When a combined fragment is initially created, the number of operands is defined by the pattern defaults. Further, you can create additional operands, or remove the existing ones.

Note that the uppermost area of the operator is empty and does not contain any operands. It is reserved for the descriptor. Clicking on this area selects the entire operator; clicking on one of the dotted rectangles selects the corresponding operand. If a combined fragment contains only one operand, the entire combined fragment and the single existing operand are still separately selectable.

See Also

OCL support overview (↗ see page 95)

UML 2.0 interaction diagrams (↗ see page 1149)

3.5.8.6.4 UML 2.0 Communication Diagram Definition

Interactions can be visually represented in your Together projects by means of the two most common interaction diagrams: Sequence and Communication.

Communication Diagrams focus on the interaction between lifelines where the architecture of the internal structure and how this corresponds with the message passing is central. The sequencing of messages is given through a sequence numbering scheme.

See Also

UML 2.0 Interaction Diagram Reference (↗ see page 1149)

Interaction (↗ see page 1150)

3.5.8.6.5 UML 2.0 Communication Diagram Elements

The table below lists the elements of UML 2.0 communication diagrams that are available using the Tool PaletteToolbox.

UML 2.0 communication diagram elements

Name	Type
Lifeline	node
Message	link

Node by Pattern	opens Pattern Wizard (see page 1162)
Link by Pattern	opens Pattern Wizard (see page 1162)
Note	annotation
Note Link	annotation link

Interaction diagrams, represented in the **Model View**, display a number of auxiliary elements that are not visible in the **Diagram View**. These elements play supplementary role for representation of the diagram structure. Actually, these elements are editable, but it is strongly advised to leave them untouched, to preserve integrity of the interaction diagrams.

3.5.8.6.6 UML 2.0 Message

Call messages are always visible in diagrams; reply messages normally are not displayed. However, you can visualize the reply message.

Messages on different diagram types

Messages in communication diagrams: When you draw a message between lifelines, a generic link line displays between the lifelines and a list of messages is created under it. The link line is present as long as there is at least one message between the lifelines.

Messages in sequence diagrams: Messages in sequence diagrams have the same properties as those in communication diagrams but allow you to perform more actions. The further discussion refers mainly to the sequence diagram messages.

Properties of the messages for both types of interaction diagrams can be edited in the Object InspectorProperties Window.

Properties of the message links

Call messages have the following properties:

Property	Description
Sequence number	Use this field to view and edit the sequential number of a message. When the message number changes, the message call changes respectively.
Name	Displays the link name. This field can be edited.
Qualified name	A read-only field that displays the fully-qualified name of the message.
Stereotype	Use this field to define the message stereotype. The stereotype name displays above the link.
Signature	Use this field to specify the name of an operation or signal associated with the message. Note that changing the signature of a message call results in changing the signature of the corresponding reply.
Arguments	Displays actual arguments of an operation associated with a message call. This field can be edited.
Sort	Use this field to select the type of synchronization from the drop-down list. The possible values are:asynchCall, synchCall, asynchSignal. The message link changes its appearance accordingly. There are certain limitations related to the asynchronous calls: Sometimes it is impossible to create or paste an asynchronous call because of the frame limitations. Execution specification for an asynchronous call must always be located on a lifeline.
Show reply message	Use this Boolean option to define whether to draw a dashed return arrow.
Commentary	Use this textual field to enter comments for a message link.

Reply messages have the following properties:

Property	Description
Stereotype	Use this field to define the message stereotype.
Attribute	Use this field to define an attribute to which the return value of the message will be assigned. This field can be edited.
Signature	Use this field to specify the name of an operation or signal associated with the message. Note that changing the signature of a message reply results in changing the signature of the corresponding call.
Arguments	Displays arguments of an operation associated with a message call. This field can be edited. Note that changing the list of arguments of a reply message results in changing the corresponding call.
Return value	Displays the return value of an operation associated with a message link. This field can be edited.
Sort	Use this field to select the type of synchronization from the drop-down list. The possible values are:asynchCall, synchCall, asynchSignal. The message link changes its appearance accordingly.
Commentary	Use this text field to comment the link.

Note: Such properties of the call and reply messages as arguments, attribute, qualified name, return value, signature, and sort pertain to the invocation specification. You can edit these properties in the invocation specification itself, in the call or in the reply messages. As a result, the corresponding properties of the counterpart message and the invocation specification will change accordingly. Stereotype and commentary properties are unique for the call and reply messages.

See Also

Working with UML 1.5 Messages (↗ see page 215)

Execution Specification and Invocation Specification (↗ see page 1151)

UML 2.0 Interaction Diagrams (↗ see page 1149)

3.5.8.6.7 UML 2.0 Sequence Diagram Definition

Interactions can be visually represented in your Together projects by means of the two most common interaction diagrams: Sequence and Communication.

The most common kind of Interaction Diagram is the Sequence Diagram, which focuses on the message interchange between a number of lifelines. A Sequence Diagram describes an interaction by focusing on the sequence of messages that are exchanged.

See Also

UML 2.0 Interaction Diagram Reference (↗ see page 1149)

Interaction (↗ see page 1150)

3.5.8.6.8 UML 2.0 Sequence Diagram Elements

The table below lists the elements of UML 2.0 sequence diagrams that are available using the Tool PaletteToolbox.

UML 2.0 sequence diagram elements

Name	Type
Lifeline	node
Execution specification	node
Combined fragment	node
State invariant	node

Message	link
Interaction use	node
Node by Pattern	opens Pattern Wizard (see page 1162)
Link by Pattern	opens Pattern Wizard (see page 1162)
Note	annotation
Note Link	annotation link

Sequence diagram can contain shortcuts to the other diagram elements. However, shortcuts to the elements that reside in the other interaction diagrams are not supported.

Interaction diagrams, represented in the **Model View**, display a number of auxiliary elements that are not visible in the **Diagram View**. These elements play supplementary role for representation of the diagram structure. Actually, these elements are editable, but it is strongly advised to leave them untouched, to preserve integrity of the interaction diagrams.

3.5.8.7 UML 2.0 State Machine Diagrams

This section describes the elements of UML 2.0 State Machine Diagrams.

Topics

Name	Description
History Element (State Machine Diagrams) (see page 1155)	The Shallow History and Deep History elements are placed on regions of the states. There may be none or one Deep History, and none or one Shallow History elements in each region. If there is only one history element in a region, it may be switched from the Deep to Shallow type by changing its kind property. Please refer to UML 2.0 Specification for more information about these elements.
UML 2.0 State Machine Diagram Definition (see page 1155)	States are the basic units of the state machines. In UML 2.0 states can have substates. Execution of the diagram begins with the Initial node and finishes with Final or Terminate node or nodes. Please refer to UML 2.0 Specification for more information about these elements.
UML 2.0 State Machine Diagram Elements (see page 1156)	The table below lists the elements of UML 2.0 State Machine diagrams that are available using the Tool PaletteToolbox. ***UML 2.0 State Machine diagram elements***

3.5.8.7.1 History Element (State Machine Diagrams)

The Shallow History and Deep History elements are placed on regions of the states.

There may be none or one Deep History, and none or one Shallow History elements in each region. If there is only one history element in a region, it may be switched from the Deep to Shallow type by changing its kind property.

Please refer to UML 2.0 Specification for more information about these elements.

See Also

Creating a history (see page 230)

State machine diagram (see page 1155)

3.5.8.7.2 UML 2.0 State Machine Diagram Definition

States are the basic units of the state machines. In UML 2.0 states can have substates.

Execution of the diagram begins with the Initial node and finishes with Final or Terminate node or nodes. Please refer to UML 2.0

Specification for more information about these elements.

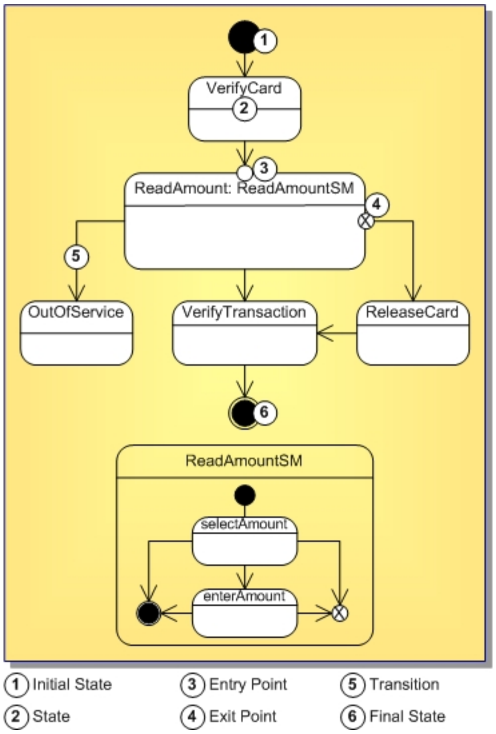
Definition

State Machine diagrams describe the logic behavior of the system, a part of the system, or the usage protocol of it.

On these diagrams you show the possible states of the objects and the transitions that cause a change in state.

State Machine diagrams in UML 2.0 are different in many aspects compared to Statechart diagrams in UML 1.5.

Sample Diagram



3.5.8.7.3 UML 2.0 State Machine Diagram Elements

The table below lists the elements of UML 2.0 State Machine diagrams that are available using the Tool PaletteToolbox.

UML 2.0 State Machine diagram elements

Name	Type	Description
State	node	
Entry point	node	Execution of the state starts at this point. It is possible to create several entry points for one state, that makes sense if there are substates.
Exit point	node	Execution of the state finishes at this point. It is possible to create several exit points for one state, that makes sense if there are substates.
Initial	node	
Final	node	
Terminate	node	
Shallow history	node	
Deep history	node	

Region	node	Use regions inside the states to group the substates. The regions may have different visibility settings and history elements. Each state has one region immediately after creation (though it can be deleted.) In the regions, you can create all the elements that are available for the State Machine diagram. only available on the state context menu
Fork	node	
Join	node	
Choice	node	
Junction	node	
Transition	link	Draw a link from the exit point of source state (or the state without exit points) to the entry point of the destination (or the state without points).
Internal transition	link	Internal transition elements are only available on the state context menu.
Node by Pattern	opens Pattern Wizard (↗ see page 1162)	
Link by Pattern	opens Pattern Wizard (↗ see page 1162)	
Note	annotation	
Note Link	annotation link	

3.5.8.8 UML 2.0 Use Case Diagrams

This section describes the elements of UML 2.0 Use Case Diagrams.

Topics

Name	Description
UML 2.0 Use Case Diagram Definition (↗ see page 1157)	Diagram courtesy of the Unified Modeling Language: *Superstructure version 2.0. August 2003. p. 536*.
UML 2.0 Use Case Diagram Elements (↗ see page 1158)	The table below lists the elements of UML 2.0 Use Case diagrams that are available using the Tool PaletteToolbox. ***UML 2.0 Use Case diagram elements***

3.5.8.8.1 UML 2.0 Use Case Diagram Definition

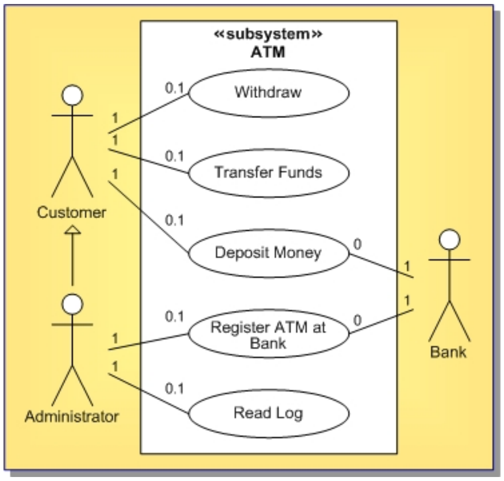
Diagram courtesy of the Unified Modeling Language: *Superstructure version 2.0. August 2003. p. 536*.

Definition

Use case diagrams describe required usages of a system, or what a system is supposed to do. The key concepts that take part in a use case diagram are actors, use cases, and subjects. A subject represents a system under consideration with which the actors and other subjects interact. The required behavior of the subject is described by the use cases.

Sample Diagram

The following diagram shows an example of actors and use cases for an ATM system.



3.5.8.8.2 UML 2.0 Use Case Diagram Elements

The table below lists the elements of UML 2.0 Use Case diagrams that are available using the Tool PaletteToolbox.

UML 2.0 Use Case diagram elements

Name	Type	Description
Actor	node	
Subject	node	
Use Case	node	
Extension point	node	Create extension points in the use cases in order to specify a point in the behavior of a use case, where this behavior can be extended by the behaviot of some other use case. This element is available on the context menu of a use case.
Extends	link	
Includes	link	
Generalization	link	
Association	link	
Node by Pattern	opens Pattern Wizard (see page 1162)	
Link by Pattern	opens Pattern Wizard (see page 1162)	
Note	annotation	
Note Link	annotation link	

3.5.9 Together Wizards

This section describes wizards used for UML modeling.

Topics

Name	Description
Create Pattern Wizard (see page 1159)	The **Create Pattern Wizard** enables you to save an existing fragment of your model as a pattern for future use. You can open the **Create Pattern Wizard** by selecting one or more nodes and links on a Class diagram and choosing the Save as Pattern command on a context menu.
New Together Project Wizards (see page 1159)	This section describes Wizards used to create new Together modeling projects.
Pattern Wizard (see page 1162)	The **Pattern Wizard** enables you to explicitly apply a pattern. You can open the **Pattern Wizard** by: • Using the **Node by Pattern** or **Link by Pattern** button in the Tool PaletteToolbox • Using the Create by Pattern command from the **Diagram View** or class context menus

3.5.9.1 Create Pattern Wizard

The **Create Pattern Wizard** enables you to save an existing fragment of your model as a pattern for future use. You can open the **Create Pattern Wizard** by selecting one or more nodes and links on a Class diagram and choosing the Save as Pattern command on a context menu.

Item	Description
File	This field specifies the target XML file name.
Name	This field specifies the name of the new pattern.
Create Pattern Object	Check this check box to use your pattern as a First Class Citizen. This means that an oval pattern element will display on your diagrams when applying the pattern.

See Also

Patterns overview (see page 96)

Creating a pattern (see page 253)

3.5.9.2 New Together Project Wizards

This section describes Wizards used to create new Together modeling projects.

Topics

Name	Description
Convert MDL Wizard (see page 1160)	Use this wizard to create a design project around an existing IBM Rational Rose (MDL) model. The wizard is invoked by the **Design Projects ▶ Convert from MDL** template of the **New Project** dialog box. **IMPORTANT:** For the MDL Import function to work, the Java Runtime Environment and the Java Development Kit must be installed, and the paths in the `jdk.config` file must correctly point to your JDK/JRE directory. For example,
Supported C# Project Wizards (see page 1161)	Together supports all C# project types available in RAD Studio. Such projects can be supplied with a UML 1.5 model. Please consult the general documentation for RAD Studio for further information about creating C# projects.
Supported Delphi Project Wizards (see page 1161)	Together supports all Delphi project types available in RAD Studio. Such projects can be supplied with a UML 1.5 model. Please consult the general documentation for RAD Studio for further information about creating Delphi projects.

UML 1.5 Together Design Project Wizard (↗ see page 1161)	**File ▶ New ▶ Other ▶ Design Projects ▶ UML 1.5 Design Project** This project is provided by Together for creation of a UML 1.5 design project. To start this wizard, choose File ▶ New ▶ Other on the main menu. The **New Items** dialog box opens. Choose the Design Projects ▶ UML 1.5 Design Project category. Click **OK**. The **New Application** dialog box opens. Specify the name and location of your model project. Click **OK**. The new UML 1.5 design project is created. Use the **Model View** to see its structure.
UML 2.0 Together Design Project Wizard (↗ see page 1161)	**File ▶ New ▶ Other ▶ Design Projects ▶ UML 2.0 Design Project** This project is provided by Together for creation of a UML 2.0 design project. To start this wizard, choose File ▶ New ▶ Other on the main menu. The **New Items** dialog box opens. Choose the Design Projects ▶ UML 2.0 Design Project category. Click **OK**. The **New Application** dialog box opens. Specify the name and location of your model project. Click **OK**. The new UML 2.0 design project is created. Use the **Model View** to see its structure.

3.5.9.2.1 Convert MDL Wizard

Use this wizard to create a design project around an existing IBM Rational Rose (MDL) model. The wizard is invoked by the Design Projects ▶ Convert from MDL template of the **New Project** dialog box.

IMPORTANT: For the MDL Import function to work, the Java Runtime Environment and the Java Development Kit must be installed, and the paths in the `jdk.config` file must correctly point to your JDK/JRE directory. For example,

```
javahome ../../../../../../../../jdk1.4.2/jre
addpath ../../../../../../../../jdk1.4.2/lib/tools.jar
```

Note that the path to the JDK/JRE should be without quotation marks.

Paths section

Button	Description
Add	Adds one model file to the Paths section. Press this button to open Select Model File dialog box, navigate to the desired model file and click Open.
Add Folder	Adds all model files in the selected folder. Press this button to open **Browse for Folder** dialog box, navigate to the desired folder that contains the model files, and click OK.
Remove	Press this button to delete the selected entry from the Paths section.
Remove all	Press this button to delete all model files from the Paths section.

Options section

Option	Description
Scale factor	Specify the element dimensions coefficient. By default, the scale factor is 0.3.
Convert Rose default colors	If this option is checked, the default Rational Rose will be replaced with the default Together colors.
Preserve diagram nodes bounds	if this option is checked, user-defined bounds are preserved in the resulting diagrams. Otherwise the default values are applied.
Convert Rose actors	This options enables you to choose mapping for the Rose classes with actor-like stereotypes (Actor, Business Actor, Business Worker, Physical Worker.) If the option is checked, the Rose actors are mapped to Together actors. If the option is not checked, the Rose actors are mapped to the classes with the Actor stereotype.

See Also

Importing a Project in IBM Rational Rose (MDL) Format (see page 265)

Project Types and Formats with Support for Modeling (see page 1116)

3.5.9.2.2 Supported C# Project Wizards

Together supports all C# project types available in RAD Studio.

Such projects can be supplied with a UML 1.5 model.

Please consult the general documentation for RAD Studio for further information about creating C# projects.

See Also

Creating a Project (see page 264)

Supported Project Formats (see page 1116)

3.5.9.2.3 Supported Delphi Project Wizards

Together supports all Delphi project types available in RAD Studio.

Such projects can be supplied with a UML 1.5 model.

Please consult the general documentation for RAD Studio for further information about creating Delphi projects.

See Also

Creating a Project (see page 264)

Supported Project Formats (see page 1116)

3.5.9.2.4 UML 1.5 Together Design Project Wizard

File ▶ New ▶ Other ▶ Design Projects ▶ UML 1.5 Design Project

This project is provided by Together for creation of a UML 1.5 design project.

To start this wizard, choose **File ▶ New ▶ Other** on the main menu. The **New Items** dialog box opens. Choose the **Design Projects ▶ UML 1.5 Design Project** category. Click **OK**.

The **New Application** dialog box opens. Specify the name and location of your model project. Click **OK**.

The new UML 1.5 design project is created. Use the **Model View** to see its structure.

See Also

Creating a project (see page 264)

Supported project formats (see page 1116)

3.5.9.2.5 UML 2.0 Together Design Project Wizard

File ▶ New ▶ Other ▶ Design Projects ▶ UML 2.0 Design Project

This project is provided by Together for creation of a UML 2.0 design project.

To start this wizard, choose **File ▶ New ▶ Other** on the main menu. The **New Items** dialog box opens. Choose the **Design**

Projects ▶ **UML 2.0 Design Project** category. Click **OK**.

The **New Application** dialog box opens. Specify the name and location of your model project. Click **OK**.

The new UML 2.0 design project is created. Use the **Model View** to see its structure.

See Also

Creating a project (see page 264)

Supported project formats (see page 1116)

3.5.9.3 Pattern Wizard

The **Pattern Wizard** enables you to explicitly apply a pattern. You can open the **Pattern Wizard** by:

- Using the **Node by Pattern** or **Link by Pattern** button in the Tool PaletteToolbox
- Using the Create by Pattern command from the **Diagram View** or class context menus

The Node by Pattern element opens the **Pattern Wizard**. You can also open the **Pattern Wizard** using the Create by Pattern command on the diagram context menu.

Item	Description
Pattern tree:	This field determines the content displayed in the Patterns pane. Use the drop-down arrow to select a pattern tree. Pattern trees are defined in the Pattern Organizer.
Panes	The following Selector/Editor panes occupy the top portion of the dialog:
Patterns:	The Patterns pane presents a tree view of the available patterns. Use the Pattern tree field to determine the available content shown in the Patterns pane.
Pattern properties:	Edit the generated class names for pattern elements, or use the information button to open the Select Element dialog for choosing elements.
Description	The Description pane lies at the bottom of the **Pattern Wizard** and displays context-sensitive help text. Help descriptions for the more complex patterns appear directly in the **Pattern Wizard** rather than in Together online help. To view a description of a pattern, click on the individual pattern in the Patterns pane.
Error	If an error occurs while applying a pattern, the Pattern Wizard remains open, and the error text displays here.
Buttons	
OK	Applies the specified pattern.
Cancel	Closes the **Pattern Wizard** without applying a pattern.

See Also

Creating model elements by pattern (see page 255)

Index

"

"gettingStarted"

"File Browser" 166

$

$ALIGN

compiler directives, Delphi 283

$APPTYPE

compiler directives, Delphi 283

$ASSERTIONS

compiler directives, Delphi 284

$AUTOBOX

compiler directives, Delphi 284

$BOOLEVAL

compiler directives (Delphi) 285

$DEBUGINFO

compiler directives, Delphi 287

$DEFINE

compiler directives, Delphi 287

$DENYPACKAGEUNIT

compiler directives, Delphi 287

$DESCRIPTION

compiler directives, Delphi 288

$DESIGNONLY

compiler directives, Delphi 288

$ELSE

compiler directives, Delphi 288

$ELSEIF

compiler directives, Delphi 289

$ENDIF

compiler directives, Delphi 289

$EXTENDEDSYNTAX

compiler directives, Delphi 290

$EXTENSION

compiler directives, Delphi 289

$EXTERNALSYM

compiler directives, Delphi 290

$FINITEFLOAT

compiler directives, Delphi 291

$HINTS

compiler directives, Delphi 291

$HPPEMIT

compiler directives, Delphi 292

$IF

compiler directives, Delphi 292

$IFDEF

compiler directives, Delphi 292

$IFEND

compiler directives, Delphi 293

$IFNDEF

compiler directives, Delphi 294

$IFOPT

compiler directives, Delphi 294

$IMAGEBASE

compiler directives, Delphi 294

$IMPLICITBUILD

compiler directives, Delphi 295

$IMPORTEDDATA

compiler directives, Delphi 295

$INCLUDE

compiler directives, Delphi 295

$IOCHECKS

compiler directives, Delphi 296

$LIB

compiler directives, Delphi 296

$LINK

compiler directives, Delphi 297

$LOCALSYMBOLS

compiler directives, Delphi 297

$LONGSTRINGS

compiler directives, Delphi 298

$MESSAGE

compiler directives, Delphi 299

$METHODINFO

compiler directives, Delphi 299

$MINENUMSIZE

compiler directives, Delphi 299

$MINSTACKSIZE; $MAXSTACKSIZE

compiler directives, Delphi 298

$NODEFINE

compiler directives, Delphi 302

$NOINCLUDE

compiler directives, Delphi 302

$ObjExportAll

compiler directives, Delphi 290

$OPENSTRINGS

compiler directives, Delphi 300

$OPTIMIZATION

compiler directives, Delphi 300

$OVERFLOWCHECKS

compiler directives, Delphi 301

$RANGECHECKS

compiler directives, Delphi 302

$REALCOMPATIBILITY

compiler directives, Delphi 302

$REFERENCEINFO; $DEFINITIONINFO

compiler directives, Delphi 305

$REGION; $ENDREGION

compiler directives, Delphi 303

$RESOURCE

compiler directives, Delphi 303

$RESOURCERESERVE

compiler directives, Delphi 310

$RUNONLY

compiler directives, Delphi 304

$SAFEDIVIDE

compiler directives, Delphi 301

$SetPEFlags

compiler directives, Delphi 310

$STACKFRAMES

compiler directives, Delphi 309

$TYPEDADDRESS

compiler directives, Delphi 305

$TYPEINFO

compiler directives, Delphi 304

$UNDEF

compiler directives, Delphi 306

$UNSAFECODE

compiler directives, Delphi 306

$VARSTRINGCHECKS

compiler directives, Delphi 306

$WARN

compiler directives, Delphi 307

$WARNINGS

compiler directives, Delphi 308

$WEAKPACKAGEUNIT

compiler directives, Delphi 308

$WRITEABLECONST

compiler directives, Delphi 309

.

.NET Assemblies 900

.NET Topics 593

.NET VCL Components 735

@

@ operator 720

<

<generic_ordered_list> Dialog Box 997

<IDname> is not a valid identifier 765

<Library Name>is already loaded, probably as a result of an incorrect program termination. Your system may be unstable and you should exit and restart Windows now. 765

A

A (Anchor) HTML Element 801

A component class named <name> already exists 763

A field or method named <name> already exists 763

About Together 83

Action List editor 921

Action Manager editor 919

Activate support 263

Activating Together Support for Projects 263

Activation Bar 1132

Active Form Wizard 771

Active Server Object wizard 772

activity diagram

sample 85

activity parameter

adding 218

Add 773

Add Address Breakpoint or Add Data Breakpoint 938

Add Comment to Event Log 1026

Add Design Packages 825

Add Exception Range 985

Add Fields 739

Add Language Exception 985

Add Languages 903

Add Namespace 57

Add New Diagram dialog box 960

Add New Project 909

Add New WebService 787, 793

Add or Edit Parameter 60

Add or Edit To-Do Item 1049

Add Page dialog box 922

Add Runtime Packages 825

Add Source Breakpoint 940

Add Symbol Table Search Path 826

Add to Project 908

Add to Repository 901, 1017, 1037

Add Watch Group 1019

Add/Edit Module Load Breakpoint 1019

Add/Remove Parameters for Operation dialog box 961

Add/Remove User Properties dialog box 961

Adding a Conditional Block 213

Adding a Member to a Container 241

Adding a Watch 116

Adding and Removing Files 153

Adding Components to a Form 152

Adding Languages to a Project 169

Adding Participants to the Patterns as First Class Citizens 253

adding references 153

Adding References 153

Adding Templates to the Object Repository 153

Advanced Data Binding 818

AJAX 24

Align fields (Delphi) 283

Align to Grid 757

Aligning Model Elements 199

Alignment 756

Annotating a Diagram 195

Another file named <FileName> is already on the search path 763

Application 826, 827

Application Module Page Options/New WebSnap Page Module 794

Application type (Delphi) 283

Apply Option Set 828

Apply Updates 985

Applying the Active Build Configuration for a Project 104

arithmetic operators 720

array parameters

parameters 672

array properties

properties 530

arrays 566

dynamic arrays 566

dynamically allocated multidimensional arrays 566

multidimensional dynamic arrays 566

static arrays 566

artifact

deploying 223

ASP .NET 825

ASP.NET 984

ASP.NET Deployment Manager 821

assembler syntax

inline assembler 610

assembly expression

inline assembler 616

Assembly Expressions (Win32 Only) 616

Assembly metadata explorer 158, 1063

Assembly Metadata Explorer 159

Assembly Metadata Explorer (Reflection viewer) 1063

assembly procedures and functions

    inline assembler 623

Assembly Procedures and Functions (Win32 Only) 623

Assembly Search Paths 734

Assert directives (Delphi) 284

Assign Local Data 739

Assigning an Element Stereotype 205

Assigning Patterns to Shortcuts 258

assignment statements 705

Associating a Lifeline with a Classifier 224

Associating a Message Link with a Method 244

Associating a Transition or a State with an Activity 230

Associating an Object with a Classifier 214

association class

    delete 239

Association Class and N-ary Association 1122

ATL 847

Attach to Process 941

Attaching to a Running Process 117

audit 98

Audit Results Pane 1111

audits

    grouping 274

    navigate to source code 274

    sorting 274

Autoboxing (Delphi for .NET) 284

AutoFormat 818

Automation Object Wizard 773

automation objects

    interfaces 633

Automation Objects (Win32 Only) 633

B

BarChart

    bar chart 98

binding

    methods 521

Block completion 42

blocks 705

blocks and scope 705

Bookmarks

    in the Code Editor 42

bookmarks, using 145

boolean operators 720

Boolean short-circuit evaluation (Delphi compiler directive) 285

boolean types

    data types 554

Branching Message Links 215

Breakpoint is set on line that contains no code or debug information. Run anyway? 765

Breakpoint List Window 1019

Breakpoints

    associating actions 118

    changing the color 118

    conditional 118

    creating a breakpoint group 118

    enabling/disabling 118

    modifying breakponts 118

    setting address breakpoint 118

    setting data breakpoints 118

    Setting module load breakpoint 118

BRIEF Keyboard Shortcuts 1069

Bring to Front 757

Browse dialog box 915

browse through sequence 92

Browse With Dialog box 774, 775

Browsing a Diagram with Overview Pane 232

Build All Projects 907

build configuations (Delphi) 5

Build configuration manager 107, 108

    Configuration manager 109, 113

Build Configuration Manager 828

Build configurations 4, 5, 6, 104

build configurations (C++) 6

Build Configurations Overview (C++) 6

Build Configurations Overview (Delphi) 5

Build configurations, active 5, 6

Build Events 829

Build output 4

Build Project 908

Building a Project Using an MSBuild Command 108

Building Packages 105

C

C++ Compiler 848, 861

C++ Compiler Advanced Compilation 849

C++ Compiler C++ Compatibility 851

C++ Compiler C++ Compilation 853

C++ Compiler Debugging 856

C++ Compiler General Compatibility 857

C++ Compiler General Compilation 858

C++ Compiler Optimizations 862

C++ Compiler Output 863

C++ Compiler Paths And Defines 864

C++ Compiler Precompiled Headers 865

C++ Compiler Warnings 866

C++ Project Options 847

Caliber

    Requirements 14

Call Stack Window 1021

calling conventions

    procedures and functions 662

calling procedures and functions

    procedures and functions 669

Calling Procedures and Functions 669

Cannot find <FileName.PAS> or <FileName.DCU> on the current search path 767

Cannot find implementation of method <MethodName> 766

case statements 705

Change 941

Change bars

    in the Code Editor 42

Change Destination File Name 777

Change Icon dialog box 916

Change Package 902

change parameters

    refactoring 66

Change Parameters 58

Change Parameters dialog box 961

Change Parameters Overview (Delphi) 66

Changes in Standard Functions and Grammar 606

Changing Appearance of Compartments 242

Changing Appearance of Interfaces 242

Changing Diagram Notation 197

Changing Type of a Link 199

character set (Delphi)

    Delphi character set 701

    reserved words 701

character types

    data types 554

chart

    creating 275

    exporting to image 275

    moving to project 275

charts

    saving 275

Choose Destination (or: Source) dialog box 962

circular unit references

    uses clause 683

class

    filtering 233

Class Completion 42

class constructor

    methods 521

class diagram

    sample 86

Class Diagram Relationships 1123

Class Diagram Types 1124

class fields

    fields 519

class helpers

    classes and objects 550

Class Helpers 550

class methods 521

    methods 521

class operators 720

class properties

  properties 530

class references

  classes and objects 539

Class References 539

class static methods

  methods 521

class types

  classes and objects 514

Class Variable in Generics 605

classes and objects 514

Classes and Objects 513, 514

classifier

  instantiating 237

Clean Package 909

clear results

  refactor delete 141

Close 799

Close function

  device drivers 692

Closing a Diagram 200

Code Browsing 42

Code Completion 42, 53

  overview 53

Code Editor 42

  customizing 141

Code Explorer 1033

code folding 137

Code Hints 42

code insight 146

Code Insight 988

  in the Code Editor 42

Code navigation 42

Code Template 96

Code visualization

  UML static diagrams 87

Code Visualization 87, 730

Code Visualization Diagram 730

Code Visualization Overview 87

CodeGear Debuggers 986

CodeGuard Configuration 980

CodeGuard Log 1059

Collection Editor 818, 922

Color 989, 1004

Color editor 917

Color Selector 815

Colors 989, 1002

Columns Collection Editor 740

COM Imports 846

COM Object Wizard 775

COM+ Event Interface Selection dialog box 776

COM+ Event Object Wizard 776

COM+ Subscription Object Wizard 776

combined fragment 1151

command line MSBuild 108

Command Line Switches and File Extensions 1082

Command Text Editor 742

CommandText Editor 741

comment 195

comment blocks 42

comments (Delphi)

  compiler directives (Delphi) 701

compartment controls

  viewing 242

Compile and Make All Projects 908

Compiler 829, 831

Compiler (Visual Basic) 836

compiler directives (Delphi) 282

Compiler directives for libraries or shared objects (Delphi) 296

Compiler Messages 835

compiling 2

Compiling and Building Procedures 104

Compiling C++ Design-Time Packages That Contain Delphi Source 106

Compiling, Building, and Running Applications 2

Component

  Import Component 733, 735

component templates 154

Components 732, 836

compound statements 705

Concepts 1

conditional block

- adding 213
- setting type 213

Conditional Block 1132

conditional compilation

- compiler directives, Delphi 285

Conditional compilation (Delphi) 285

Configuration Levels 1088

Configuration manager 107, 108

Configure Cassini 824

Configure Data Adapter 742

Configure Tools 1008

Configure Virtual Directory 824

Configuring the Documentation Generation Facility 247

Configuring the Memory Manager 175

Configuring Together 183

Connection Editor 743

Connection String Editor (ADO) 744

constants

- data types 589

Constraints Collection Editor 740

Constraints in Generics 603

constructors

- Delphi for .NET 653
- methods 521

Convert MDL Wizard 1160

Converting Between UML 1.5 Sequence and Collaboration Diagrams 215

Copy 758

Copying and Pasting an Execution or Invocation Specification 224

Copying and Pasting Model Elements 200

Copying and Pasting Shortcuts, Folders or Pattern Trees 258

Copying References to a Local Path 154

Could not stop due to hard mode 763

CPU Window 1022

Create Component Template 732

Create Pattern Wizard 1159

Creating a Browse-Through Sequence of Diagrams 246

Creating a Component Template 154

Creating a Deferred Event 235

Creating a Delegation Connector 221

Creating a Diagram 196

Creating a Folder in the Pattern Organizer 259

Creating a Guard Condition for a Transition 230

Creating a History Element 230

Creating a Link by Pattern 255

Creating a Link with Bending Points 207

Creating a Member for a State 231

Creating a Metrics Chart 275

Creating a Model Element by Pattern 255

Creating a Multiple Transition 235

Creating a Pattern 253

Creating a Pin 217

Creating a Port 222

Creating a Project 155, 264

Creating a Referenced Part 221

Creating a Self-Transition 236

Creating a Sequence or Communication Diagram from an Interaction 225

Creating a Shortcut 208

Creating a Shortcut to a Pattern 259

Creating a Simple Link 209

Creating a Single Model Element 209

Creating a State 231

Creating a State Invariant 225

Creating a Virtual Pattern Tree 259

Creating an Activity for a State 211

Creating an Association Class 239

Creating an Extension Point 246

Creating an Inner Classifier 239

Creating an Internal Structure for a Node 221

Creating an Internal Transition 235

Creating an OCL Constraint 249

Creating Build Events 107

Creating Live Templates 138

Creating Multiple Elements 207

Creating Named Build Configurations for C++ 107

Creating Named Build Configurations for Delphi 108

Creating Template Libraries 138

creating templates

  live templates 138

Creation Order 756

custom attributes 593

Customize New Menu 777

Customize Toolbars 1033

Customizing Code Editor 141

Customizing the Form 155

Customizing the Tool Palette 156

Customizing Toolbars 156

Cut 758

D

Data Adapter Dataset 744

Data Breakpoint is set on a stack location 762

Data Explorer 1034

data types, variables, and constants 553

Data Types, Variables, and Constants 552, 553

DataAdapter Preview 744

database

  adding a new connection 165

Database 737

Database Editor 745

Database Form Wizard 745

Databindings 818

Dataset Properties 746

DCC32 869

DDE Info dialog box 918

Debug information (Delphi) 287

Debug Inspector 942

Debug session in progress. Terminate? 767, 948

Debug Windows 1018

Debugger 836

Debugger Environment Block 839

Debugger Exception Notification 943

Debugger Options 990

Debugger Symbol Tables 845

debugging

  adding a watch 116

  attaching to a process 117

  breakpoints 118

  inspecting data elements 122

  modifying expressions 124

  overview 10

  preparation 121, 124

  remote applications 125

Debugging Applications 10

Debugging Procedures 114

Debugging VCL for .NET Source Code 121

Declaration of class <ClassName> is missing or incorrect 764

declarations 705

declarations and statements 705

Declarations and Statements 705

declare field 62

  refactoring 143

Declare Field 61

declare variable

  initial type 62

  refactoring 143

Declare Variable 64

Declare Variable and Declare Field Overview (Delphi) 62

declare variable and field 62

declare variable and field samples 62

declare variable rules 62

Declared Constants 589

declaring exception types

  exceptions 541

Declaring Generics 598

declaring types

  data types 586

Declaring Types 586

decomposition

  defining 226

Default Keyboard Shortcuts 1073

Default Options 909

default parameters

    parameters 672

Deferred Event 1117

DEFINE directive (Delphi) 287

Delegation Connector 1146

delegation, implementing interfaces by

    interfaces 627

Delete 758

Delete Saved Desktop 1036

Delete Templates 914

deleting

    references 141

Deleting a Diagram 200

Deleting Patterns as First Class Citizens from the Model 254

Deleting shortcuts, folders or pattern trees 260

Delphi Compiler 874

Delphi Compiler Compiling 870

Delphi compiler directives 282

Delphi Compiler Directives (List) 280

Delphi Compiler Errors 311

Delphi Compiler Other Options 875

Delphi Compiler Paths and Defines 876

Delphi Compiler Warnings 877

Delphi Language Guide 512

Delphi language overview 657

Delphi Overview 656

Delphi packages (C++) 106

Delphi Reference 280

Delphi Runtime Errors 509

DENYPACKAGEUNIT directive (Delphi) 287

Deploying Applications 19, 133

Deploying ASP.NET applications 133

Deploying the AdoDbx Client 134

deployment

    overview 19

deployment diagram

    sample 85, 86

Deployment manager

    ASP.NET 821

Description 837

Description (Delphi) 288

Design Pattern 96

Designing a UML 1.5 Activity Diagram 211

Designing a UML 1.5 Component Diagram 212

Designing a UML 1.5 Deployment Diagram 212

Designing a UML 1.5 Statechart Diagram 217

Designing a UML 2.0 Activity Diagram 218

Designing a UML 2.0 Component Diagram 220

Designing a UML 2.0 Deployment Diagram 223

Designing a UML 2.0 Sequence or Communication Diagram 226

Designing a UML 2.0 State Machine Diagram 232

Designing Use Case Hierarchy 246

Designing User Interfaces 15

DESIGNONLY directive (Delphi) 288

Desktop layouts 161

Desktop Toolbar 1036

Desktops 1059

destructors

    methods 521

Detach From Program 949

Developing Tests 180

development lifecycle

    tool overview 14

device drivers 692

diagram

    renaming 204

Diagram Layout Algorithms 963

Diagram Layout Overview 92

Diagram View 1106

    opening 267

Directives (Delphi) 701

Directories/Conditionals 838

Disabling Themes in the IDE and in Your Application 157

Disassembly 1032

dispatch interface methods

    automation objects 633

Display 992

Displaying Expanded Watch Information 117

Dispose

Delphi for .NET 653

DisposeCount cannot be declared in classes with destructors 336

DIV HTML Element 803

DLLs, calling 635

doc comment 93

Dock Edit Window 1060

docking

tools 157

Docking Tool Windows 157

documentation generation 100

Documentation Generation Facility Overview 100

Driver Settings 746

DUnit Overview 72

dynamic properties 161

Dynamic Properties 818

dynamically loaded libraries

libraries 637

E

E1038: Unit identifier '%s' does not match file name 495

E2001: Ordinal type required 466

E2002: File type not allowed here 394

E2003: Undeclared identifier: '%s' 407

E2004: Identifier redeclared: '%s' 407

E2005: '%s' is not a type identifier 491

E2006: PACKED not allowed here 471

E2007: Constant or type identifier expected 368

E2008: Incompatible types 363

E2009: Incompatible types: '%s' 364

E2010: Incompatible types: '%s' and '%s' 365

E2011: Low bound exceeds high bound 434

E2012: Type of expression must be BOOLEAN 441

E2013: Type of expression must be INTEGER 444

E2014: Statement expected, but expression of type '%s' found 452

E2015: Operator not applicable to this operand type 507

E2016: Array type required 440

E2017: Pointer type required 447

E2018: Record, object or class type required 449

E2019: Object type required 446

E2020: Object or class type required 446

E2021: Class type required 441

E2022: Class helper type required 443

E2023: Function needs result type 450

E2024: Invalid function result type 484

E2025: Procedure cannot have a result type 455

E2026: Constant expression expected 367

E2027: Duplicate tag value 381

E2028: Sets may have at most 256 elements 486

E2029: %s expected but %s found 390

E2030: Duplicate case label 381

E2031: Label expected 445

E2032: For loop control variable must have ordinal type 398

E2033: Types of actual and formal var parameters must be identical 505

E2034: Too many actual parameters 490

E2035: Not enough actual parameters 439

E2036: Variable required 451

E2037: Declaration of '%s' differs from previous declaration 399

E2038: Illegal character in input file: '%s' (%s) 408

E2045: Bad object file format: '%s' 346

E2049: Label declaration not allowed in interface part 430

E2050: Statements not allowed in interface part 350

E2052: Unterminated string 498

E2053: Syntax error in real number 348

E2054: Illegal type in Write/Writeln statement 350

E2055: Illegal type in Read/Readln statement 348

E2056: String literals may have at most 255 elements 488

E2057: Unexpected end of file in comment started on line %ld 498

E2058: Class, interface and object types only allowed in type section 464

E2059: Local class, interface or object types not allowed 465

E2060: Class and interface types only allowed in type section 360

E2061: Local class or interface types not allowed 360

E2062: Virtual constructors are not allowed 465

E2064: Left side cannot be assigned to 462

E2065: Unsatisfied forward or external declaration: '%s' 400

E2066: Missing operator or semicolon 438

E2067: Missing parameter type 439

E2068: Illegal reference to symbol '%s' in object file '%s' 412

E2070: Unknown directive: '%s' 495

E2071: This type cannot be initialized 354

E2072: Number of elements (%d) differs from declaration (%d) 420

E2073: Label already defined: '%s' 431

E2074: Label declared and referenced, but not set: '%s' 431

E2075: This form of method call only allowed in methods of derived types 445

E2076: This form of method call only allowed for class methods 441

E2078: Procedure FAIL only allowed in constructor 391

E2079: Procedure NEW needs constructor 452

E2080: Procedure DISPOSE needs destructor 380

E2081: Assignment to FOR-Loop variable '%s' 340

E2082: TYPEOF can only be applied to object types with a VMT 451

E2083: Order of fields in record constant differs from declaration 482

E2085: Unit name mismatch: '%s' '%s' 440

E2086: Type '%s' is not yet completely defined 418

E2088: Variable name expected 504

E2089: Invalid typecast 425

E2090: User break - compilation aborted 502

E2091: Segment/Offset pairs not supported in CodeGear 32-bit Pascal 485

E2093: Label '%s' is not declared in current procedure 458

E2094: Local procedure/function '%s' assigned to procedure variable 433

E2095: Missing ENDIF directive 437

E2096: Method identifier expected 436

E2097: BREAK or CONTINUE outside of loop 351

E2098: Division by zero 381

E2099: Overflow in conversion or arithmetic operation 339

E2100: Data type too large: exceeds 2 GB 492

E2101: Size of data type is zero 492

E2102: Integer constant too large 422

E2103: 16-Bit fixup encountered in object file '%s' 396

E2104: Bad relocation encountered in object file '%s' 348

E2105: Inline assembler syntax error 345

E2106: Inline assembler stack overflow 345

E2107: Operand size mismatch 343

E2108: Memory reference expected 342

E2109: Constant expected 341

E2110: Type expected 346

E2111: Cannot add or subtract relocatable symbols 344

E2112: Invalid register combination 344

E2113: Numeric overflow 343

E2114: String constant too long 345

E2115: Error in numeric constant 342

E2116: Invalid combination of opcode and operands 340

E2117: 486/487 instructions not enabled 340

E2118: Division by zero 341

E2119: Structure field identifier expected 341

E2120: LOOP/JCXZ distance out of range 430

E2121: Procedure or function name expected 448

E2122: PROCEDURE or FUNCTION expected 473

E2123: PROCEDURE, FUNCTION, PROPERTY, or VAR expected 360

E2124: Instance member '%s' inaccessible here 392

E2125: EXCEPT or FINALLY expected 389

E2126: Cannot BREAK, CONTINUE or EXIT out of a FINALLY clause 356

E2127: 'GOTO %s' leads into or out of TRY statement 401

E2128: %s clause expected, but %s found 361

E2129: Cannot assign to a read-only property 478

E2130: Cannot read a write-only property 479

E2131: Class already has a default property 374

E2132: Default property must be an array property 373

E2133: TYPEINFO standard function expects a type identifier 493

E2134: Type '%s' has no type info 457

E2136: No definition for abstract method '%s' allowed 337

E2137: Method '%s' not found in base class 468

E2138: Invalid message parameter list 427

E2139: Illegal message method index 412

E2140: Duplicate message method index 385

E2142: Inaccessible value 502

E2143: Expression has no value 390

E2144: Destination is inaccessible 376

E2145: Re-raising an exception only allowed in exception handler 483

E2146: Default values must be of ordinal, pointer or small set type 375

E2147: Property '%s' does not exist in base class 493

E2148: Dynamic method or message handler not allowed here 476

E2149: Class does not have a default property 442

E2150: Bad argument type in variable type array constructor 349

E2151: Could not load RLINK32.DLL 440

E2152: Wrong or corrupted version of RLINK32.DLL 507

E2153: ';' not allowed before 'ELSE' 485

E2154: Type '%s' needs finalization - not allowed in variant record 395

E2155: Type '%s' needs finalization - not allowed in file type 395

E2156: Expression too complicated 489

E2157: Element 0 inaccessible - use 'Length' or 'SetLength' 501

E2158: %s unit out of date or corrupted: missing '%s' 349

E2159: %s unit out of date or corrupted: missing '%s.%s' 349

E2160: Type not allowed in OLE Automation call 346

E2163: Too many conditional symbols 490

E2165: Compile terminated by user 502

E2166: Unnamed arguments must precede named arguments in OLE Automation call 496

E2167: Abstract methods must be virtual or dynamic 337

E2168: Field or method identifier expected 443

E2169: Field definition not allowed after methods or properties 391

E2170: Cannot override a non-virtual method 469

E2171: Variable '%s' inaccessible here due to optimization 504

E2172: Necessary library helper function was eliminated by linker (%s) 402

E2173: Missing or invalid conditional symbol in '$%s' directive 426

E2174: '%s' not previously declared as a PROPERTY 429

E2175: Field definition not allowed in OLE automation section 391

E2176: Illegal type in OLE automation section: '%s' 410

E2177: Constructors and destructors not allowed in OLE automation section 368

E2178: Dynamic methods and message handlers not allowed in OLE automation section 388

E2179: Only register calling convention allowed in OLE automation section 461

E2180: Dispid '%d' already used by '%s' 383

E2181: Redeclaration of property not allowed in OLE automation section 474

E2182: '%s' clause not allowed in OLE automation section 408

E2183: Dispid clause only allowed in OLE automation section 378

E2184: %s section valid only in class types 412

E2185: Overriding automated virtual method '%s' cannot specify a dispid 411

E2186: Published Real property '%s' must be Single, Real, Double or Extended 480

E2187: Size of published set '%s' is >4 bytes 481

E2188: Published property '%s' cannot be of type %s 347

E2189: Thread local variables cannot be local to a function 434

E2190: Thread local variables cannot be ABSOLUTE 336

E2191: EXPORTS allowed only at global scope 390

E2192: Constants cannot be used as open array arguments 367

E2193: Slice standard function only allowed as open array argument 487

E2194: Cannot initialize thread local variables 419

E2195: Cannot initialize local variables 418

E2196: Cannot initialize multiple variables 419

E2197: Constant object cannot be passed as var parameter 368

E2198: %s cannot be applied to a long string 404

E2199: Packages '%s' and '%s' both contain unit '%s' 470

E2200: Package '%s' already contains unit '%s' 470

E2201: Need imported data reference ($G) to access '%s' from unit '%s' 470

E2202: Required package '%s' not found 439

E2203: $WEAKPACKAGEUNIT '%s' contains global data 507

E2204: Improper GUID syntax 435

E2205: Interface type required 444

E2206: Property overrides not allowed in interface type 475

E2207: '%s' clause not allowed in interface type 409

E2208: Interface '%s' already implemented by '%s' 425

E2209: Field declarations not allowed in interface type 392

E2210: '%s' directive not allowed in in interface type 376

E2211: Declaration of '%s' differs from declaration in interface '%s' 423

E2212: Package unit '%s' cannot appear in contains or uses clauses 502

E2213: Bad packaged unit format: %s.%s 471

E2214: Package '%s' is recursively required 483

E2215: 16-Bit segment encountered in object file '%s' 485

E2216: Can't handle section '%s' in object file '%s' 496

E2217: Published field '%s' not a class or interface type 354

E2218: Published method '%s' contains an unpublishable type 355

E2220: Never-build package '%s' requires always-build package '%s' 458

E2221: $WEAKPACKAGEUNIT '%s' cannot have initialization or finalization code 506

E2222: $WEAKPACKAGEUNIT & $DENYPACKAGEUNIT both specified 354

E2223: $DENYPACKAGEUNIT '%s' cannot be put into a package 354

E2224: $DESIGNONLY and $RUNONLY only allowed in package unit 412

E2225: Never-build package '%s' must be recompiled 470

E2226: Compilation terminated; too many errors 490

E2227: Imagebase is too high - program exceeds 2 GB limit 413

E2228: A dispinterface type cannot have an ancestor interface 377

E2229: A dispinterface type requires an interface identification 378

E2230: Methods of dispinterface types cannot specify directives 377

E2231: '%s' directive not allowed in dispinterface type 409

E2232: Interface '%s' has no interface identification 423

E2233: Property '%s' inaccessible here 477

E2234: Getter or setter for property '%s' cannot be found 437

E2236: Constructors and destructors must have %s calling convention 460

E2237: Parameter '%s' not allowed here due to default value 372

E2238: Default value required for '%s' 371

E2239: Default parameter '%s' must be by-value or const 374

E2240: $EXTERNALSYM and $NODEFINE not allowed for '%s'; only global symbols 487

E2241: C++ obj files must be generated (-jp) 370

E2242: '%s' is not the name of a unit 462

E2245: Recursive include file %s 482

E2246: Need to specify at least one dimension for SetLength of dynamic array 339

E2247: Cannot take the address when compiling to byte code 338

E2248: Cannot use old style object types when compiling to byte code 464

E2249: Cannot use absolute variables when compiling to byte code 336

E2250: There is no overloaded version of '%s' that can be called with these arguments 454

E2251: Ambiguous overloaded call to '%s' 338

E2252: Method '%s' with identical parameters already exists 386

E2253: Ancestor type '%s' does not have an accessible default constructor 438

E2254: Overloaded procedure '%s' must be marked with the 'overload' directive 447

E2255: New not supported for dynamic arrays - use SetLength 502

E2256: Dispose not supported (nor necessary) for dynamic arrays 453

E2257: Duplicate implements clause for interface '%s' 384

E2258: Implements clause only allowed within class types 415

E2259: Implements clause only allowed for properties of class or interface type 415

E2260: Implements clause not allowed together with index clause 413

E2261: Implements clause only allowed for readable property 416

E2262: Implements getter must be %s calling convention 415

E2263: Implements getter cannot be dynamic or message method 413

E2264: Cannot have method resolutions for interface '%s' 414

E2265: Interface '%s' not mentioned in interface list 416

E2266: Only one of a set of overloaded methods can be published 386

E2267: Previous declaration of '%s' was not marked with the 'overload' directive 448

E2268: Parameters of this type cannot have default values 373

E2270: Published property getters and setters must have %s calling convention 462

E2271: Property getters and setters cannot be overloaded 467

E2272: Cannot use reserved unit name '%s' 489

E2273: No overloaded version of '%s' with this parameter list exists 455

E2274: property attribute 'label' cannot be used in dispinterface 379

E2275: property attribute 'label' cannot be an empty string 477

E2276: Identifier '%s' cannot be exported 354

E2277: Only external cdecl functions may use varargs 505

E2278: Cannot take address of local symbol %s 355

E2279: Too many nested conditional directives 452

E2280: Unterminated conditional directive 498

E2281: Type not allowed in Variant Dispatch call 350

E2282: Property setters cannot take var parameters 486

E2283: Too many local constants. Use shorter procedures 489

E2284: Duplicate resource name: type %d '%s' 388

E2285: Duplicate resource id: type %d id %d 388

E2286: Coverage library name is too long: %s 371

E2287: Cannot export '%s' multiple times 440

E2288: File name too long (exceeds %d characters) 393

E2289: Unresolved custom attribute: %s 497

E2290: Cannot mix destructors with IDisposable 376

E2291: Missing implementation of interface method %s.%s 423

E2292: '%s' must reference a property or field of class '%s' 478

E2293: Cannot have both a DLLImport attribute and an external or calling convention directive 381

E2294: A class helper that descends from '%s' can only help classes that are descendents '%s' 428

E2295: A class helper cannot introduce a destructor 402

E2296: A constructor introduced in a class helper must call the parameterless constructor of the helped class as the first statement 428

E2297: Procedure definition must be ILCODE calling convention 350

E2298: read/write not allowed for CLR events. Use Include/Exclude procedure 427

E2299: Property required 449

E2300: Cannot generate property accessor '%s' because '%s' already exists 474

E2301: Method '%s' with identical parameters and result type already exists 384

E2302: 'Self' is uninitialized. An inherited constructor must be called before accessing ancestor field '%s' 371

E2303: 'Self' is uninitialized. An inherited constructor must be called before calling ancestor method '%s' 371

E2304: 'Self' is uninitialized. An inherited constructor must be called 370

E2305: 'Self' might not have been initialized 371

E2306: 'Self' is initialized more than once 370

E2307: NEW standard function expects a dynamic array type identifier 339

E2308: Need to specify at least one dimension for NEW of dynamic array 339

E2309: Attribute - Known attribute named argument cannot be an array 352

E2310: Attribute - A custom marshaler requires the custom marshaler type 352

E2311: Attribute - MarshalAs fixed string requires a size 352

E2312: Attribute - Invalid argument to a known attribute 352

E2313: Attribute - Known attribute cannot specify properties 352

E2314: Attribute - The MarshalAs attribute has fields set that are not valid for the specified unmanaged type 352

E2315: Attribute - Known custom attribute on invalid target 352

E2316: Attribute - The format of the GUID was invalid 353

E2317: Attribute - Known custom attribute had invalid value 353

E2318: Attribute - The MarshalAs constant size cannot be negative 353

E2319: Attribute - The MarshalAs parameter index cannot be negative 353

E2320: Attribute - The specified unmanaged type is only valid on fields 353

E2321: Attribute - Known custom attribute has repeated named argument 353

E2322: Attribute - Unexpected type in known attribute 353

E2323: Attribute - Unrecognized argument to a known custom attribute 353

E2324: Attribute - Known attribute named argument doesn't support variant 353

E2325: Attribute '%s' is not valid on this target 357

E2326: Attribute '%s' can only be used once per target 357

E2327: Linker error while emitting attribute '%s' for '%s' 352

E2328: Linker error while emitting metadata 496

E2329: Metadata - Error occured during a read 358

E2330: Metadata - Error occured during a write 358

E2331: Metadata - File is read only 358

E2332: Metadata - An ill-formed name was given 358

E2333: Metadata - Data value was truncated 359

E2334: Metadata - Old version error 358

E2335: Metadata - A shared mem open failed to open at the originally 359

E2336: Metadata - Create of shared memory failed. A memory mapping of the same name already exists 359

E2337: Metadata - There isn't .CLB data in the memory or stream 359

E2338: Metadata - Database is read only 359

E2339: Metadata - The importing scope is not compatible with the emitting scope 358

E2340: Metadata - Data too large 358

E2341: Metadata - Column cannot be changed 358

E2342: Metadata - Too many RID or primary key columns, 1 is max 358

E2343: Metadata - Primary key column may not allow the null value 358

E2344: Metadata - Data too large 359

E2345: Metadata - Attempt to define an object that already exists 436

E2346: Metadata - A guid was not provided where one was required 436

E2347: Metadata - Bad binary signature 436

E2348: Metadata - Bad input parameters 435

E2349: Metadata - Cannot resolve typeref 436

E2350: Metadata - No logical space left to create more user strings 436

E2351: Final methods must be virtual or dynamic 395

E2352: Cannot override a final method 469

E2353: Cannot extend sealed class '%s' 391

E2354: String element cannot be passed to var parameter 488

E2355: Class property accessor must be a class field or class static method 361

E2356: Property accessor must be an instance field or method 476

E2357: PROCEDURE, FUNCTION, or CONSTRUCTOR expected 473

E2358: Class constructors not allowed in class helpers 357

E2359: Multiple class constructors in class %s: %s and %s 440

E2360: Class constructors cannot have parameters 358

E2361: Cannot access private symbol %s.%s 481

E2362: Cannot access protected symbol %s.%s 480

E2363: Only methods of descendent types may access protected symbol [%s]%s.%s across assembly boundaries 472

E2364: Cross-assembly protected reference to [%s]%s.%s in %s.%s 432

E2366: Global procedure or class static method expected 450

E2370: Cannot use inherited methods for interface property accessors 474

E2371: ABSTRACT and FINAL cannot be used together 337

E2372: Identifier expected 406

E2373: Call to abstract method %s.%s 336

E2374: Cannot make unique type from %s 354

E2375: PRIVATE or PROTECTED expected 472

E2376: STATIC can only be used on non-virtual class methods 430

E2377: Unable to locate Borland.Delphi.Compiler.ResCvt.dll 484

E2378: Error while converting resource %s 389

E2379: Virtual methods not allowed in record types 506

E2380: Instance or class static method expected 444

E2381: Resource string length exceeds Windows limit of 4096 characters 484

E2382: Cannot call constructors using instance variables 422

E2383: ABSTRACT and SEALED cannot be used together 338

E2385: Error while signing assembly 389

E2386: Invalid version string '%s' specified in %s 430

E2387: The key container name '%s' does not exist 428

E2388: Unrecognized strong name key file '%s' 428

E2389: Protected member '%s' is inaccessible here 480

E2390: Class must be sealed to call a private constructor without a type qualifier 481

E2391: Potentially polymorphic constructor calls must be virtual 462

E2392: Can't generate required accessor method(s) for property %s.%s due to name conflict with existing symbol %s in the same scope 355

E2393: Invalid operator declaration 429

E2394: Parameterless constructors not allowed on record types 472

E2395: Unsafe procedure only allowed if compiling with {$UNSAFECODE ON} 498

E2396: Unsafe code only allowed in unsafe procedure 497

E2397: Unsafe pointer only allowed if compiling with {$UNSAFECODE ON} 498

E2398: Class methods in record types must be static 482

E2399: Namespace conflicts with unit name '%s' 381

E2400: Unknown Resource Format '%s' 496

E2401: Failure loading .NET Framework %s: %08X 361

E2402: Constructing instance of abstract class '%s' 370

E2403: Add or remove accessor for event '%s' cannot be found 438

E2404: Cannot mix READ/WRITE property accessors with ADD/REMOVE accessors 440

E2405: Unknown element type found importing signature of %s.%s 496

E2406: EXPORTS section allowed only if compiling with {$UNSAFECODE ON} 497

E2407: Duplicate resource identifier %s found in unit %s(%s)

and %s(%s) 388

E2408: Can't extract strong name key from assembly %s 489

E2409: Fully qualified nested type name %s exceeds 1024 byte limit 452

E2410: Unsafe pointer variables, parameters or consts only allowed in unsafe procedure 498

E2411: Unit %s in package %s refers to unit %s which is not found in any package. Packaged units must refer only to packaged units 460

E2412: CREATE expected 370

E2413: Dynamic array type needed 388

E2414: Disposed_ cannot be declared in classes with destructors 381

E2415: Could not import assembly '%s' because it contains namespace '%s' 430

E2416: Could not import package '%s' because it contains system unit '%s' 430

E2417: Field offset cannot be determined for variant record because previous field type is unknown size record type 496

E2418: Type '%s' needs initialization - not allowed in variant record 420

E2419: Record type too large: exceeds 1 MB 482

E2420: Interface '%s' used in '%s' is not yet completely defined 417

E2422: Imported identifier '%s' conflicts with '%s' in namespace '%s' 361

E2423: Void type not usable in this context 506

E2424: Codepage '%s' is not installed on this machine 426

E2425: Inline methods must not be virtual nor dynamic 421

E2426: Inline function must not have asm block 421

E2427: Only one of IID or GuidAttribute can be specified 407

E2428: Field '%s' needs initialization - not allowed in CLS compliant value types 420

E2429: Duplicate implementation for 'set of %s' in this scope 388

E2430: for-in statement cannot operate on collection type '%s' 463

E2431: for-in statement cannot operate on collection type '%s' because '%s' does not contain a member for '%s', or it is inaccessible 362

E2432: %s cannot be applied to a rectangular dynamic array 429

E2433: Method declarations not allowed in anonymous record or local record type 437

E2434: Property declarations not allowed in anonymous record or local record type 476

E2435: Class member declarations not allowed in anonymous record or local record type 360

E2436: Type declarations not allowed in anonymous record or local record type 491

E2437: Constant declarations not allowed in anonymous record or local record type 370

E2439: Inline function must not have open array argument 466

E2441: Inline function declared in interface section must not use local symbol '%s' 422

E2442: Inline directive not allowed in constructor or destructor 421

E2447: Duplicate symbol '%s' defined in namespace '%s' by '%s' and '%s' 385

E2448: An attribute argument must be a constant expression, typeof expression or array constructor 346

E2449: Inlined nested routine '%s' cannot access outer scope variable '%s' 421

E2450: There is no overloaded version of array property '%s' that can be used with these arguments 454

E2452: Unicode characters not allowed in published symbols 494

E2453: Destination cannot be assigned to 376

E2454: Slice standard function not allowed for VAR nor OUT argument 487

Edit 754

Edit Hyperlinks for Diagram dialog box 963

Edit Object Info 1008

Edit Page dialog box 923

Edit Tab Order 756

Edit Tools 1008

Editing an OCL Expression 249

editing code

- class completion 145
- code insight 146
- code snippets 148
- collapsing code 137

Editing Code Procedures 136

Editing Properties 260

Editing Resource Files in the Translation Manager 170

Editor Options 993

ELSE (Delphi) 288

ELSEIF (Delphi) 289

ENDIF directive 289

Enter Address to Position 954

Enter New Value 1025

Enter Search Bytes 1025

entry and exit action

  creating 217

enumerated types

  data types 554

Environment Options 994

Environment Variables 995

Epsilon Keyboard Shortcuts 1076

Error address not found 762

Error creating process: <Process> (<ErrorCode>) 763

Error Messages 311, 760

Establishing a Connection for Remote Debugging 127

Evaluate

  Modify/debugging 10

Evaluate/Modify 944

Event Log Options 995

Event Log Window 1025

event properties and event handlers

  events 536

events

  classes and objects 536

Events 536

exceptions

  classes and objects 541

Exceptions 541

Executable extension (Delphi) 289

Execution Specification and Invocation Specification 1151

Exit 799

expandable tooltips 122

Explorer 996

Exploring .NET Assembly Metadata Using the Reflection Viewer 158

Exploring Windows Type Libraries 159

export clauses

  libraries 637

Export Diagram to Image 731

Export Diagram to Image dialog box 964

Export symbols (Delphi) 290

Export Visual Studio Project 1009

Exporting a Diagram to an Image 197

Exporting a Pattern 257

Exporting a Project to XMI Format 264

Exporting Audit Results 271

expressions 720

Expressions 720

Extended syntax (Delphi) 290

Extension Point 1137

external declarations

  procedures and functions 662

External Symbols (Delphi) 290

Extract Interface or Superclass dialog box 964

extract method

  refactoring 60

Extract Method 64

Extract Method dialog box 965

Extract Method Overview (Delphi) 60

extract resource string

  converting strings 143

Extract Resource String 66

Extract Resource String (Delphi) 62

F

F1027: Unit not found: '%s' or binary equivalents (%s) 394

F2039: Could not create output file '%s' 393

F2040: Seek error on '%s' 394

F2046: Out of memory 436

F2047: Circular unit reference to '%s' 359

F2048: Bad unit format: '%s' 495

F2051: Unit %s was compiled with a different version of %s.%s 505

F2063: Could not compile used unit '%s' 502

F2069: Line too long (more than 1023 characters) 432

F2084: Internal Error: %s%d 422

F2087: System unit incompatible with trial version 376

F2092: Program or unit '%s' recursively uses itself 482

F2220: Could not compile package '%s' 470

F2438: UCS-4 text encoding not supported. Convert to UCS-2 or UTF-8 430

F2446: Unit '%s' is compiled with unit '%s' in '%s' but different version '%s' found 376

fatal errors 511

Fatal errors 511

Field <Field Name> does not have a corresponding component. Remove the declaration? 764

Field <Field Name> should be of type <Type1> but is declared as <Type2>. Correct the declaration? 764

Field Link Designer 746

fields

- classes and objects 519

Fields 519

Fields Editor 746

File 768

File Browser 1036

File Extensions of Files Generated by RAD Studio 1084

file input and output

- standard routines 692

file types (Win32) 566

Filter editor 918

Filter To-Do List 1050

finalization section 683

Finalize

- Delphi for .NET 653

Find 952

Find Class 955

Find in Files 953

Find Local References 956

Find Option 878

Find Original Symbol 956

Find Package Import 944

Find Reference Results 1060

find references

- find declaration symbol 65
- find local references 141

Find References 954, 956

Find References Overview (Delphi, C#, C++) 65

find references sample 65

Find Unit 66

Finding Items on the Tool Palette 158

Finding References 141

Finding Units and Using Namespaces (Delphi, C#) 142

Flip Children 758

Floating Point Exception Checking (Delphi) 291

Flush function

- device drivers 692

Folder or Directory View 867

Font 1004

Font editor 919

for loops 705

for...in statements 705

Foreign Key Constraint 747

Form Designer 46, 1004

Forms 840

forward and interface declarations

- procedures and functions 662

foward declarations

- classes and objects 514

FPU 1026

FTP Connection Options 778

function calls

- expressions 720

function declarations

- procedures and functions 662

Fundamental Syntactic Elements 700, 701

G

General Concepts 14

Generate Dataset 748

Generate Documentation dialog box 966

Generate Sequence Diagram dialog box 966

Generating an Incremental Sequence Diagram 245

Generating Project Documentation 248

Generics

- constraints 603
- overview 596
- parameterized methods 598
- Parameterized types 596
- platform requirements 596

procedural types 598

scope of type params 598

terminology 597

Generics (Parameterized Types) 595

getting started

adding components to forms 152

adding files to a project 153

adding templates to repository 153

arranging toolbars 156

copying references 154

creating projects 155

customizing forms 155

customizing tool palette 156

docking tool windows 157

filtering searched components 158

installing custom components 160

renaming files 160

saving desktop layouts 161

setting component properties 161

setting project options 162, 163

setting properties and events 164

setting tool preferences 165

Vista themes 157

writing event handlers 168

Getting Started 22

Getting Started Procedures 151

Getting Started with Together 83

Go to Line Number 954

goto statements 705

Grouping Actions into an Activity 219

guard expression 1118

GUI Components for Modeling 1105

H

H2077: Value assigned to '%s' never used 503

H2135: FOR or WHILE loop executes zero times - deleted 463

H2164: Variable '%s' is declared but never used in '%s' 499

H2219: Private symbol '%s' declared but never used 472

H2235: Package '%s' does not use or export '%s.%s' 470

H2244: Pointer expression needs no Initialize/Finalize - need ^ operator? 480

H2365: Override method %s.%s should match case of ancestor %s.%s 468

H2368: Visibility of property accessor method %s should match property %s.%s 474

H2369: Property accessor %s should be %s 474

H2384: CLS: overriding virtual method '%s.%s' visibility (%s) must match base class '%s' (%s) 362

H2440: Inline method visibility is not lower or same visibility of accessing member '%s.%s' 435

H2443: Inline function '%s' has not been expanded because unit '%s' is not specified in USES list 421

H2444: Inline function '%s' has not been expanded because accessing member '%s' is inaccessible 421

H2445: Inline function '%s' has not been expanded because its unit '%s' is specified in USES statement of IMPLEMENTATION section and current function is inline function or being inline function 421

H2451: Narrowing given WideChar constant (#$%04X) to AnsiChar lost information 371

H2455: Narrowing given wide string constant lost information 371

help

.NET Framework 51

borland site 51

Delphi for .NET 51

quick start guide 51

typographic 51

Help Insight 42, 1060

Help on Help 51

Hiding and Showing Model Elements 232

hinting directives

compiler directives,, delphi 705

Hints (Delphi) 291

History Element (State Machine Diagrams) 1155

History Manager 149, 1009

HPP emit (Delphi) 292

HR HTML Element 804

HTML Elements 801

HTML Formatting 998

HTML Tidy Options 999

HTML/ASP.NET Options 998

Hyperlinking Diagrams 201

I

I
- O errors 510

I/O Errors 510

IBDatabase Editor dialog box 752

IBTransaction Editor dialog box 753

IBUpdateSQL and IBDataSet Editor dialog box 753

IconView Items editor 924

ide
- Code Editor 34
- component tray 46
- design guidelines 46
- design surface 34
- form preview 46
- forms 34
- HTML designer 46
- MS Active Accessibility 34
- Object Inspector 34
- object repository 34
- Project Manager 34
- tool palette 34
- visual components 46
- welcome page 34

IDE Classic Keyboard Shortcuts 1070

IDE Command Line Switches and Options 1082

IDE on Windows Vista 40

identifiers (Delphi)
- character set (Delphi) 701

IF directive (Delphi) 292

if statements 705
- with statements 705

IFDEF directive (Delphi) 292

IFEND directive (Delphi) 293

IFNDEF directive (Delphi) 294

IFOPT directive (Delphi) 294

IInterface 625

iLink 32 878

Image base address 294

Image List Editor 925

IMG HTML Element 805

IMPLEMENTATION part is missing or incorrect 765

implementation section 683

implementing interfaces
- interfaces 627

Implementing Interfaces 627

Implib 886

Implicit Build (Delphi) 295

import
- namespaces 143

Import Component 733

Imported data 295

Importing a Legacy Pattern 257

Importing a Project Created in TVS, TEC, TJB, or TPT 265

Importing a Project in IBM Rational Rose (MDL) Format 265

Importing a Project in XMI Format 266

importing functions
- procedures and functions 662

Include file (Delphi) 295

Incorrect field declaration in class <ClassName> 763

Incorrect method declaration in class <ClassName> 765

Increasing the Memory Address Space 176

Incremental Search 956

index specifiers
- properties 530

indexes
- expressions 720

Information 904

inheritance and scope
- classes and objects 514

inherited methods
- methods 521

initial value
- defining 243

initialization section 683

inline assembler 610

Inline Assembly Code (Win32 Only) 609

inline directive 669

Inline Variable dialog box 967

inner classifier

- creating 239

Inner Classifiers 1124

InOut function

- device drivers 692

INPUT HTML Element 806

Input Mask editor 929

Input output checking (Delphi) 296

Insert 813

Insert Image 813

Insert Input 814

Insert Object dialog box 930

Insert Table 814

Insert Template 915

Insert User Control 813

Inspect 945

Inspecting and Changing the Value of Data Elements 122

Installed .NET Components 734

Installing a Debugger on a Remote Machine 126

Installing Custom Components 160

Installing More Computer Languages 110

Installing, Starting, and Stopping the Remote Debug Server 125

instance specification

- defining features 237
- instantiating 237

Instantiating a Classifier 211

Insufficient memory to run 765

integer types

- data types 554

interaction

- navigating 226

Interaction 1150

interaction diagram

- return link 215

interaction diagrams

- navigation 267

interaction use 1150

- creating 226

interface 1143

- changing notation 242
- hiding 240

interface references

- interfaces 631

Interface References 631

interface section 683

Interface Selection Wizard 778

Interfaces

- GUID 625

Internal Data Formats 645

internal errors 130

Introduce Field dialog box 968

Introduce Variable dialog box 968

Invalid event profile <Name> 768

K

Key Mappings 1068

Keyboard Mappings 1068

keystroke macros 142

KiviatChart

- kiviat chart 98

L

labels (Delphi) 701

Language Exceptions 999

Language Overview 657

Laying Out a Diagram Automatically 202

layout

- setting up 202

Librarian 886

libraries

- initialization code 637

libraries and packages 635

Libraries and Packages 634, 635

Library 991

lifeline

- associating 226

lifeline type

associating 226
Link object file (Delphi) 297
Linker 840, 880
Linker error: %s 484
Linker error: %s: %s 485
Linker Linking 878
Linker Output Options 881
Linker Warnings 884
Linking Another Interaction from an Interaction Diagram 227
Linking Delphi Units Into an Application 111
linking to object files
procedures and functions 662
List Editor 885
ListView Items Editor 926
Live Templates 42
LiveSource Overview 93
LiveSource Rules 1125
Load Picture dialog box 931
Load Process 949
Load Process Environment Block 945
Load Process Local 946
Load Process Remote 946
Load Process Symbol Tables 947
Load String List dialog box 932
Loading an image at design time 931
local declarations
procedures and functions 662
Local symbol information (Delphi) 297
Local Variables Window 1028
Localization Procedures 169
Localizing Applications 18
Lock Controls 758
logical (bitwise) operators 720
long strings
string types 561
Long strings (Delphi) 298
loops 705

M

macro
recording 142
Managing Memory 175
Managing the Development Cycle Overview 14
Masked Text editor 932
MDA 93
Members 1125
Memory 1032
Memory allocation sizes (Delphi) 298
memory management
Delphi for .NET 653
Delphi for Win32 644
Memory Management 644
Memory Management Issues on the .NET Platform 653
Memory Management on the Win32 Platform 644
Menus 1110
message
operation call 227
self 227
MESSAGE directive (Delphi) 299
message methods
methods 521
message pane
refactoring 111
Message View 1038
method extract
code fragments 143
method rename 59
METHODINFO directive (Delphi) 299
methods
classes and objects 521
Methods 521
metric 98
viewing description 276
Metric Results Pane 1112
metrics
filtering results 276

navigating 276

sorting results 276

updating results 276

Minimum enumeration size (Delphi) 299

Misaligned Data Breakpoint 762

model

reloading 269

Model Annotation Overview 91

model element

hiding and showing 232

navigating 267

opening 267

Model Element Overview 91

Model Hyperlinking Overview 92

Model Import and Export Overview 100

Model Shortcut Overview 92

Model Support 969

Model View 1112

Model View Window 1058

Modeling Applications with Together 81

Modeling Overview 89

Modifying Variable Expressions 124

Module header is missing or incorrect 764

Modules Window 1028

Monitoring Memory Usage 177

Move dialog box 969

Moving Model Elements 203

MSBuild 4

MSBuild command 108

MSBuild Overview 4

multicast events

events 536

Multi-line Editor 1054

N

name mapping 94

Named Option Sets Overview 7

namespace

deleting 250

opening 250

renaming 250

viewing 250

Namespace and Package Overview 89

namespaces 689

programs and units 689

namespaces, declaring 689

namespaces, multi-unit 689

namespaces, searching 689

Native OS Exceptions 999

Nested Constants

Nested Types 546

nested exceptions

exceptions 541

Nested Message 1133

Nested Type Declarations 546

nested types

classes and objects 546

NET_VS 817

New 800

New Application 782

New ASP.NET Application 779

New ASP.NET Content Page 779

New ASP.NET Generic Handler 779

New ASP.NET Master Page 780

New ASP.NET User Control 780

New ASP.NET Web Service 780

New Category Name 904

New Component 735

New Connection 748

New Console Application 780

New DBWeb Control Wizard 781

New Dynamic-link Library 781

New Edit Window 1057

New Expression 948

New Field 748

New Items 781

New Remote Data Module Object 782

New SOAP Server Application 788

New Standard Action Classes dialog box 927

New Tags 1000

New Thread Object 783

New Together Project Wizards 1159

New VCL Component Wizard 736

New Web Server Application 793

New WebSnap Application 795

New WebSnap Data Module 796

No code was generated for the current line 766

node

- optimizing size 204
- optimizing size (global) 204

NODEFINE 302

NOINCLUDE (Delphi) 302

Notebook editor 933

null-terminated string functions

- strings 692

null-terminated strings

- strings 692

numerals (Delphi)

- character set (Delphi) 701

NUnit Overview 76

O

Object Inspector 1000, 1038, 1113

object interfaces

- interfaces 625

Object Interfaces 624, 625

Object Repository 1010

OCL 95

- constraint 95
- expression 95
- supported diagram types 95

OCL object constraint

- creating 249

OCL Support Overview 95

online help

- using 248

Open 784

open array parameters

- parameters 672

open arrays 669

Open dialog box 933

Open function

- device drivers 692

Open String Parameters (Delphi) 300

Opening an Existing Project for Modeling 267

Opening the Pattern Organizer 260

Opening the UML 2.0 Sample Project 190

operand 1151

operating system errors 512

Operating system errors 512

operation

- defining parameters 223

operator 1151

Operator and Operand for a Combined Fragment 1151

operator overloading

- classes and objects 548

Operator Overloading 548

operator precedence 720

operators

- classes and objects 539
- expressions 720

Optimization (Delphi) 300

option sets (C++) 7

Option Value Editors 1101

options

- disabling changes 183

Options 905, 909

ordinal types

- data types 554

Overflow checking (Delphi) 301

overloading methods

- methods 521

Overloads and Type Compatibility in Generics 602

Override System Variable/New User Variable/Edit User Variable 1000

Overview of Debugging 10

Overview of Generics 596

Overview of Remote Debugging 12
Overview of Virtual Folders 50

P

Package 784
packages 640
  compiling 105, 640
  units 105
Packages 640, 734, 843, 1005
parameters
  procedures and functions 669, 672
Parameters 672, 950
parameters, passing 679
Paste 758
Paste Special dialog box 934
Paths and Defines 887
Paths and Directories (C++) 987
pattern
  stub implementation 255
Pattern GUI Components 1107
Pattern Organizer 96, 1107
Pattern Registry 96, 1109
Pattern Wizard 1162
Patterns as First Class Citizens 96
  adding participants 253
Patterns Overview 96
PE (portable executable) header flags (Delphi) 310
Pentium-safe FDIV operations (Delphi) 301
Picture editor 934
pin
  creating 217
Pin 1141
pointer operators 720
pointer types
  data types 575
Pointers and Pointer Types 575
Pre-Build, Pre-Link, or Post-Build Events 844
Preparing a Project for Debugging 124
Preparing Files for Remote Debugging 128
Previewing and Applying Refactoring Operations 111
Print Audit dialog box 970
Print Diagram dialog box 971
Print Selection 785
Printing a Diagram 197
Printing Audit Results 272
procedural types
  data types 578
Procedural Types 578
procedure declarations
  procedures and functions 662
Procedures 103
procedures and functions 662
  overloading 662
Procedures and Functions 662
program control 679
Program Control 678, 679
program heading 683
program organization
  programs and units 683
Program Reset 950
program structure 683
Programs and Units 682, 683
project
  importing 265
Project 819
  Clean 1038
Project Dependencies 903
Project Manager 1038
Project Options 822, 842
Project Page Options 904
Project Properties 888
Project References 901
Project Types and Formats with Support for Modeling 1116
Project Updated 785
Project Upgrade 785
projects 155
  additional projects 47
  types of 47

Propeditors 910

properties

classes and objects 530

Properties 530, 819

property access 530

Property and method <MethodName> are not compatible 767

property overrides

properties 530

Pull Members Up and Push Members Down dialog boxes 972

Q

QA Audits dialog window 972

QA Metrics dialog window 974

Quality Assurance Facilities Overview 98

Quality Assurance GUI Components 1110

R

RAD Studio Dialogs and Commands 730

raising and handling

exceptions 541

Range checking 302

real types

data types 554

Real48 compatibility (Delphi) 302

Recording a Keystroke Macro 142

records

advanced 566

traditional 566

Redo 759

refactoring 57

applying 111

extract resource string 62

preview 111

procedures 143

symbol rename 143

Refactoring

change parameters 184

extract interface 185

extract method 185

extract superclass 186

in the Code Editor 42

inline variable 186

introduce variable 187

introdule field 187

move members 188

Pull Up/Push Down 188

rename 189

safe delete 189

Refactoring Applications 55

Refactoring Code 143

Refactoring Overview 57, 98

Refactoring: "Safe Delete" 189

Refactoring: “Pull Members Up" and “Push Members Down” 188

Refactoring: Changing Parameters 184

Refactoring: Creating Inline Variables 186

Refactoring: Extracting Interfaces 185

Refactoring: Extracting Method 185

Refactoring: Extracting Superclass 186

Refactoring: Introducing Fields 187

Refactoring: Introducing Variables 187

Refactoring: Moving Members 188

Refactoring: Renaming Elements 189

Refactorings 67

Reference 279

referenced project 270

references

projects 153

Reflection viewer 158

Regions (Delphi and C#) 303

Register ActiveX Server 950

Registering Memory Leaks 178

Registers 1032

Relation 749

relational operators 720

Relations Collection Editor 740

Remote Data Module Wizard 786

Remote Debugging: Metaprocedure 125

Remove from Project 904

Remove Language 903

Rename 975

Rename <symbol name> (C#) 68

Rename <symbol name> (Delphi) 69

Rename Connection 749

rename symbol 112

- preview 111
- refactoring 112

Rename Symbol (C++) 68

Renaming a Diagram 204

Renaming a Symbol 112

Renaming Files Using the Project Manager 160

repeat loops 705

Replace Text 955

Repository 1006

required interface

- creating 238

re-raising exceptions

- exceptions 541

Rerouting a Link 204

Reserved address space for resources (Delphi) 310

reserved words (Delphi) 701

Resizing Model Elements 204

Resolving Internal Errors 130

Resource Compiler 867

Resource Compiler Options 868

Resource Compiler Paths And Defines 869

Resource file (Delphi) 303

Revert to Previous Revision 787

role

- binding to different classifiers 222

Run 936, 950

Run To Cursor and Run Until Return 950

Running Audits 273

Running Metrics 276

RUNONLY directive (Delphi) 304

runtime errors 509

Runtime type information (Delphi) 304

S

Safe Delete dialog box 975

Satellite Assembly Wizard 787

Save As 788

Save Audit and Metric Results dialog box 976

Save Desktop 1047

Save Picture As dialog box 935

Save String List dialog box 936

Saving Changes in the Pattern Registry 261

Saving Desktop Layouts 161

Scale 757

scope 705

Search 952, 1066

Search Again 955

Search for Usages dialog box 977

Searching Diagrams 209

Searching Source Code for Usages 210

Select All 759

Select All Controls 759

Select Debug Desktop 1047

Select Directory 789

Select element dialog box 977

SELECT HTML Element 808

Select Icon 906

Select Menu 915

Selecting Model Elements 205

Selection Manager 978

Self identifier

- methods 521

Send to Back 759

Set Active Language 903

set constructors

- expressions 720

set operators 720

sets 98, 566

- using 274, 277

Setting and Modifying Breakpoints 118

Setting C++ Project Options 163

Setting Component Properties 161
Setting Dynamic Properties 161
Setting Project Options 162
Setting Properties and Events 164
Setting the Active Language for a Project 171
Setting The IDE To Mimic Delphi 7 164
Setting the Search Order for Debug Symbol Tables 129
Setting Tool Preferences 165
Setting Up the External Translation Manager 172
Sharing Memory 178
Sharing Patterns 258
short strings
- string types 561

shortcut
- create 208

shortcut to classifier
- creating 214

shortcuts
- other 1104

Show Borders 1060
Show Execution Point 951
Show Grid 1061
Show Tag Glyphs 1061
Showing and Hiding an OCL Constraint 250
Signing 844
simple types
- data types 554

Simple Types 554
Size 757
slot
- adding 237
- associating 237

slot stereotype
- defining 237

slot value
- setting 237

Snap To Grid 1061
SOAP Data Module Wizard 788
Sort Fields Editor 750
Sorting Patterns 261
source breakpoint 118
source control
- basics 16
- files 16
- projects 16
- repository basics 16

Source Control Options 1001
Source File Not Found 1030
Source has been modified. Rebuild? 767
Source Options 1001
SPAN HTML Element 809
Specifying Entry and Exit Actions 236
SQL Monitor 749
Stack 1032
Stack frames (Delphi) 309
standard exception classes and routines
- exceptions 541

standard routines
- System unit 692

Standard Routines and I/O 692
Starting a Project 47
state
- nested 234

State 1117
state invariant 1150
- connecting 225

State Machine diagram
- sample 85

statements 705
- simple statements 705

static methods
- methods 521

Step Over 951
stereotype
- assigning 205
- defining 205

storage specifiers
- properties 530

Stored Procedures Dialog 751

strict visibility

- classes and objects 514

String List editor 927

string operators 720

string parameters

- parameters 672

string types

- data types 561

String Types 561

strings 701

Structure View 1047

structured types

- data types 566

Structured Types 566

stub implementation 96

- Node by Pattern 255
- using 255

subrange types

- data types 554

Supported C# Project Wizards 1161

Supported Delphi Project Wizards 1161

Supported UML Specifications 90

Symbol <BrowseSymbol> not found. 767

Symbol declaration and cross-reference information (Delphi) 305

symbol rename

- refactoring 59

Symbol Rename Overview (Delphi, C#, C++) 59

symbols (Delphi)

- character set (Delphi) 701

Sync Edit 150

- editing code 67

Sync Edit Mode (Delphi, C#, C++) 67

SyncEdit 42

Synchronizing the Model View, Diagram View, and Source Code 267

syntax (Delphi)

- character set (Delphi) 701

Syntax Check for Project 910

System macros 1103

T

TABLE HTML Element 810

Table Properties 1050

TableMappings Collection Editor 751

Tables Collection Editor 741

targets files 8

Targets files 8

tasm32 889

template libraries 42

Template Libraries 50, 138, 1010

- project 138
- projects 50

templates 138, 148

Templates Window 1048

Terminology for Generics 597

test case

- writing tests 180

test project

- Adding source to a test project 180

Testing Applications 70

Testing Wizards 815

Text file type

- file types 692

TEXTAREA HTML Element 811

The <Method Name> method referenced by <Form Name> does not exist. Remove the reference? 766

The <Method Name> method referenced by <Form Name>.<Event Name> has an incompatible parameter list. Remove the reference? 766

The project already contains a form or module named <Name> 763

themes 157

Threads 1030

tie frame 1150

TObject and TClass

- classes and objects 514

To-Do List 1049

to-do lists

- overview 42

planning 166

Together 957

Together Configuration Options 1086

Together Diagram Appearance Options 1089

Together Diagram Layout Options 1091

Together Diagram Overview 90

Together Diagram Print Options 1094

Together Diagram Procedures 190

Together Diagram View Filters Options 1096

Together Documentation Generation Procedures 247

Together General Options 1098

Together Generate Documentation Options 1099

Together Glossary 1139

Together Keyboard Shortcuts 1104

Together Model View Options 1101

Together Object Constraint Language (OCL) Procedures 248

Together Option Categories 1088

Together Options dialog window 969

Together Pattern Procedures 251

Together Procedures 183

Together Project Overview 89

Together Project Procedures 262

Together Quality Assurance Procedures 271

Together Refactoring Operations 1115

Together Refactoring Procedures 184

Together Reference 1086

Together Sequence Diagram Roundtrip Options 1102

Together Source Code Options 1103

Together Wizards 1158

Toggle Form/Unit 1058

Tool Palette 1003, 1051, 1114

components 152

toolbars

customizing 156

Toolbars 1061

Tools 979

Tools Options 982

Tools Overview 41

Tools Properties 1011

Tour of the IDE 34

Trace Into 951

Trace to Next Source Line 951

Transactional Object Wizard 789

transform to source code 94

Transformation to Source Code Overview 94

Transforming a Design Project to Source Code 269

transition

multiple 1118

self 236

Transition 1118

Translation Editor 1062

Translation Manager 1052

Translation Repository 1006

translation tools

adding languages to a project 169

editing with Translation Manager 170

External Translation Manager 173

overview 18

setting the active language 171

setting up the External Translation Manager 172

updating resource modules 173

Translation Tools Options 1005

TreeView Items Editor 927

Troubleshooting a Model 269

try... finally statements

exceptions 541

try...except statements

exceptions 541

Turbo Assembler 889

Turbo Assembler Options 889

Turbo Assembler Paths and Defines 891

Turbo Assembler Warnings 892

Type Cast 948

type compatibility

data types 583

Type Compatibility and Identity 583

Type Library (C++) 988

Type Library (Delphi) 1007

Type Library Editor 1054

Type Library Explorer 1065

typecasts

expressions 720

Type-checked pointers (Delphi) 305

U

UDDI Browser 902

UML 90

UML 1.5 Activity Diagram Definition 1119

UML 1.5 Activity Diagram Elements 1120

UML 1.5 Activity Diagrams 1117

UML 1.5 Class Diagram Definition 1126

UML 1.5 Class Diagram Elements 1127

UML 1.5 Class Diagrams 1121

UML 1.5 Collaboration Diagram Definition 1133

UML 1.5 Component Diagram Definition 1128

UML 1.5 Component Diagram Elements 1129

UML 1.5 Component Diagrams 1128

UML 1.5 Deployment Diagram Definition 1130

UML 1.5 Deployment Diagram Elements 1131

UML 1.5 Deployment Diagrams 1129

UML 1.5 Interaction Diagram Elements 1134

UML 1.5 Interaction Diagrams 1131

UML 1.5 Message 1133

UML 1.5 Reference 1116

UML 1.5 Sequence Diagram Definition 1134

UML 1.5 Statechart Diagram Definition 1135

UML 1.5 Statechart Diagram Elements 1136

UML 1.5 Statechart Diagrams 1135

UML 1.5 Together Design Project Wizard 1161

UML 1.5 Use Case Diagram Definition 1137

UML 1.5 Use Case Diagram Elements 1138

UML 1.5 Use Case Diagrams 1137

UML 2.0 Activity Diagram Definition 1141

UML 2.0 Activity Diagram Elements 1142

UML 2.0 Activity Diagrams 1140

UML 2.0 Class Diagram Definition 1143

UML 2.0 Class Diagram Elements 1144

UML 2.0 Class Diagrams 1143

UML 2.0 Communication Diagram Definition 1152

UML 2.0 Communication Diagram Elements 1152

UML 2.0 Component Diagram Definition 1145

UML 2.0 Component Diagram Elements 1145

UML 2.0 Component Diagrams 1145

UML 2.0 Composite Structure Diagram Definition 1146

UML 2.0 Composite Structure Diagram Elements 1147

UML 2.0 Composite Structure Diagrams 1146

UML 2.0 Deployment Diagram Definition 1148

UML 2.0 Deployment Diagram Elements 1148

UML 2.0 Deployment Diagrams 1148

UML 2.0 Interaction Diagrams 1149

UML 2.0 Message 1153

UML 2.0 Reference 1140

UML 2.0 Sample Project 84

UML 2.0 Sample Project, Behavior Package 85

UML 2.0 Sample Project, Structure Package 86

UML 2.0 Sequence Diagram Definition 1154

UML 2.0 Sequence Diagram Elements 1154

UML 2.0 State Machine Diagram Definition 1155

UML 2.0 State Machine Diagram Elements 1156

UML 2.0 State Machine Diagrams 1155

UML 2.0 Together Design Project Wizard 1161

UML 2.0 Use Case Diagram Definition 1157

UML 2.0 Use Case Diagram Elements 1158

UML 2.0 Use Case Diagrams 1157

UML in color

using 198

UML In Color 90

Unavailable Options 893

UNDEF directive (Delphi) 306

Understanding Assembler Syntax (Win32 Only) 610

undo

refactoring 67

Undo 759

Undoing a Refactoring (Delphi, C#) 67

Unique Constraint 752

Unit 803

unit names 689

unit structure 683

Unit Test Case Wizard 816

Unit Test Procedures 180

Unit Test Project Wizard 816, 817

unit testing 70

Unit Testing Overview 70

unit tests 180

units

- Delphi for .NET 653
- linking 111
- namespaces 142

unmanaged code 47

Unregister ActiveX Server 951

Unsafe Code (Delphi for .NET) 306

untyped files

- file types 692

Update Localized Projects 910

Updating Resource Modules 173

Use Unit 790

uses clause 683

- unit references 683

Uses clause is missing or incorrect 767

Using .NET Custom Attributes 593

Using a Class Diagram as a View 240

Using Bookmarks 145

Using Class Completion 145

Using Code Folding 137

Using Code Insight 146

Using Design Guidelines with VCL Components 165

Using Drag-and-Drop 206

Using Grid and Other Appearance Options 198

Using Inline Assembly Code (Win32 Only) 610

Using Live Templates 148

Using Namespaces with Delphi 689

Using Online Help 248

Using Source Control 16

Using Sync Edit 150

Using Targets Files 109

Using the CPU View 116

Using the External Translation Manager 173

Using the File Browser 166

Using the History Manager 149

Using the Pattern Organizer 261

Using the Pattern Registry 254

Using the Stub Implementation Pattern 255

Using the UML in Color Profile 198

Using To-Do Lists 166

Using Tooltips During Debugging 122

Using View Filters 233

Using Virtual Folders 167

V

Value List editor 928

Value typecasts

- expressions 720

variables

- data types 587

Variables 587

variant parts in 566

variant types

- data types 580

Variant Types 580

Var-string checking (Delphi) 306

VCL Designer 990

Version Info 845

View 1014

View Form 1057

View Source 910

View Units 1057

Viewing Audit Results 274

Viewing Metric Results 276

virtual and dynamic methods

- methods 521

Virtual folders 167

visibility

- classes and objects 514

visibility modifier

defining 243

Vista themes 34

Visual Basic Keyboard Shortcuts 1078

Visual Studio Keyboard Shortcuts 1079

W

W1000: Symbol '%s' is deprecated 406

W1001: Symbol '%s' is specific to a library 406

W1002: Symbol '%s' is specific to a platform 407

W1003: Symbol '%s' is experimental 406

W1004: Unit '%s' is specific to a library 495

W1005: Unit '%s' is specific to a platform 495

W1006: Unit '%s' is deprecated 494

W1007: Unit '%s' is experimental 494

W1009: Redeclaration of '%s' hides a member in the base class 404

W1010: Method '%s' hides virtual method of base type '%s' 403

W1011: Text after final 'END.' - ignored by compiler 401

W1013: Constant 0 converted to NIL 509

W1014: String constant truncated to fit STRING[%ld] 487

W1015: FOR-Loop variable '%s' cannot be passed as var parameter 397

W1016: Typed constant '%s' passed as var parameter 492

W1017: Assignment to typed constant '%s' 340

W1018: Case label outside of range of case expression 356

W1021: Comparison always evaluates to False 366

W1022: Comparison always evaluates to True 366

W1023: Comparing signed and unsigned types - widened both operands 365

W1024: Combining signed and unsigned types - widened both operands 362

W1029: Duplicate %s '%s' with identical parameters will be inacessible from C++ 382

W1031: Package '%s' will not be written to disk because -J option is enabled 470

W1032: Exported package threadvar '%s.%s' cannot be used outside of this package 471

W1034: $HPPEMIT '%s' ignored 405

W1035: Return value of function '%s' might be undefined 456

W1036: Variable '%s' might not have been initialized 499

W1037: FOR-Loop variable '%s' may be undefined after loop 396

W1039: No configuration files found 453

W1040: Implicit use of Variants unit 417

W1041: Error converting Unicode char to locale charset. String truncated. Is your LANG environment variable set correctly? 494

W1042: Error converting locale string '%s' to Unicode. String truncated. Is your LANG environment variable set correctly? 434

W1043: Imagebase $%X is not a multiple of 64k. Rounding down to $%X 413

W1044: Suspicious typecast of %s to %s 489

W1045: Property declaration references ancestor private '%s.%s' 472

W1046: Unsafe type '%s%s%s' 497

W1047: Unsafe code '%s' 497

W1048: Unsafe typecast of '%s' to '%s' 497

W1049: value '%s' for option %s was truncated 466

W1050: WideChar reduced to byte char in set expressions 507

W1051: Duplicate symbol names in namespace. Using '%s.%s' found in %s. Ignoring duplicate in %s 388

W1052: Can't find System.Runtime.CompilerServices.RunClassConstructor. Unit initialization order will not follow uses clause order 495

W1053: Local PInvoke code has not been made because external routine '%s' in package '%s' is defined using package local types in its custom attributes 432

W1055: Published caused RTTI ($M+) to be added to type '%s' 492

W1201: XML comment on '%s' has badly formed XML--'Whitespace is not allowed at this location.' 509

W1202: XML comment on '%s' has badly formed XML--'Reference to undefined entity '%s'' 509

W1203: XML comment on '%s' has badly formed XML--'A name was started with an invalid character.' 508

W1204: XML comment on '%s' has badly formed XML--'A name contained an invalid character.' 508

W1205: XML comment on '%s' has badly formed XML--'The character '%c' was expected.' 508

W1206: XML comment on '%s' has cref attribute '%s' that could not be resolved 508

W1207: XML comment on '%s' has a param tag for '%s', but there is no parameter by that name 508

W1208: Parameter '%s' has no matching param tag in the XML comment for '%s' (but other parameters do) 508

Warning messages (Delphi) 307

Warnings (Delphi) 308

Watch List Window 1031

Watch Properties 948

Weak packaging 308

Web App Components 796

Web App Debugger 1013

Web Deploy Options 906

WebSnap 1007

Welcome Page 1062

What is RAD Studio? 23

What's New in RAD Studio (C++Builder) 28

What's New in RAD Studio (Delphi for Microsoft .NET) 24

What's New in RAD Studio (Delphi) 32

What's New in Together 88

when to use exceptions

- exceptions 541

while loops 705

wide-character strings 692

WideString

- string types 561

Win View 1062

Window List 1057

with statements 705

Working with a Collaboration Use 222

Working with a Combined Fragment 228

Working with a Complex State 234

Working with a Constructor 243

Working with a Field 243

Working with a Namespace or a Package 250

Working with a Provided or Required Interface 238

Working with a Referenced Project 270

Working with a Relationship 241

Working with a Set of Audits 274

Working with a Set of Metrics 277

Working with a Tie Frame 229

Working with a UML 1.5 Message 215

Working with a UML 2.0 Message 227

Working with an Instance Specification 237

Working with an Interface 240

Working with an Object Flow or a Control Flow 219

Working with Named Option Sets 113

Working with User Properties 206

Writeable typed constants (Delphi) 309

Writing Dynamically Loaded Libraries 637

Writing Event Handlers 168

WSDL Import Options 791

WSDL Import Wizard 792

X

x1008: Integer and HRESULT interchanged 405

x1012: Constant expression violates subrange bounds 351

x1019: For loop control variable must be simple local variable 398

x1020: Constructing instance of '%s' containing abstract method '%s.%s' 369

x1025: Unsupported language feature: '%s' 498

x1026: File not found: '%s' 393

x1028: Bad global symbol definition: '%s' in object file '%s' 346

x1030: Invalid compiler directive: '%s' 426

x1033: Unit '%s' implicitly imported into package '%s' 417

x1054: Linker error: %s 436

x2041: Read error on '%s' 394

x2042: Write error on '%s' 395

x2043: Close error on '%s' 393

x2044: Chmod error on '%s' 393

x2141: Bad file format: '%s' 393

x2243: Expression needs no Initialize/Finalize 491

x2269: Overriding virtual method '%s.%s' has lower visibility (%s) than base class '%s' (%s) 459

x2367: Case of property accessor method %s.%s should be %s.%s 474

x2421: Imported identifier '%s' conflicts with '%s' in '%s' 361

XMI Export dialog box 978

XMI Import dialog box 979

XML Data Binding Wizard Options 797

XML Data Binding Wizard, page 1 797

XML Data Binding Wizard, page 2 798

XML Data Binding Wizard, page 3 799

XML Mapper 1011

Z

Zooming a Diagram 234