Adobe® Creative Suite 2

Bridge JavaScript Reference

© 2005 Adobe Systems Incorporated. All rights reserved.

Adobe® Creative Suite 2 Bridge JavaScript Reference for Windows® and Macintosh®.

NOTICE: All information contained herein is the property of Adobe Systems Incorporated. No part of this publication (whether in hardcopy or electronic form) may be reproduced or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of Adobe Systems Incorporated. The software described in this document is furnished under license and may only be used or copied in accordance with the terms of such license.

This publication and the information herein is furnished AS IS, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies, makes no warranty of any kind (express, implied, or statutory) with respect to this publication, and expressly disclaims any and all warranties of merchantability, fitness for particular purposes, and noninfringement of third party rights.

Any references to company names in sample templates are for demonstration purposes only and are not intended to refer to any actual organization.

Adobe, the Adobe logo, Acrobat, GoLive, Illustrator, Photoshop, InDesign, and Version Cue are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Apple, Mac, Macintosh, and Mac OS are trademarks of Apple Computer, Inc., registered in the United States and other countries. Microsoft, and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and other countries. JavaScript and all Java-related marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. UNIX is a registered trademark of The Open Group.

All other trademarks are the property of their respective owners.

If this guide is distributed with software that includes an end user agreement, this guide, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. Except as permitted by any such license, no part of this guide may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Adobe Systems Incorporated. Please note that the content in this guide is protected under copyright law even if it is not distributed with software that includes an end user license agreement.

The content of this guide is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Contents

Welcome **19**
About This Book 19
Who should read this book 19
What is in this book 19
Document conventions 20
Typographical conventions 20
JavaScript conventions 20
Where to go for more information 21

1 Scripting Bridge **22**
Scripting Overview 22
Executing Scripts 22
Communicating with Other Applications 23
The Bridge Document Object Model 23
The application and documents 24
Thumbnails in documents 25
Thumbnails as node references 25
Using and accessing thumbnails 25
User interface objects 26
Navigation bars 26
Dialogs 26
Menus 27
Events 27
Application preferences 27
The Bridge DOM and the Bridge Browser Window 28

2 Event Handling and Script-Defined Browse Schemes **31**
Event Handling in Bridge 31
Defining event handlers 31
Registering event handlers 32
Event handling examples 32
User-interface events 35
Script-Defined Browse Schemes 36
Defining and registering a browse scheme 36

3 Creating a User Interface **38**
User Interface Options for Scripts 38
Navigation Bars 38
Dialogs Boxes 39
Content Pane 39
ScriptUI User Interfaces 40
Displaying ScriptUI Dialogs 40
Displaying ScriptUI elements in a navigation bar 40
Displaying HTML in Bridge 42
Defining callbacks for HTML scripts 42
Executing script functions defined on HTML UI pages 43
Displaying HTML in Bridge Dialogs 44

Communicating with Bridge from dialog JavaScript 44
Using callbacks in an HTML dialog 45
Calling functions defined in an HTML dialog 47
Displaying HTML in a Navigation Bar 47
Using callbacks from an HTML navigation bar 47
Calling functions defined in an HTML navigation bar 48
Displaying HTML in the Content Pane 50
Callback example: Requesting specific metadata value for a file 50
Passing Complex Values in Remote Calls 51
Passing an object from Bridge to HTML/JavaScript 52
Scheduling tasks from callbacks 53
Scheduling a remote function execution 54

4 Using File and Folder Objects 56

Overview 56
Specifying Paths 56
Absolute and relative path names 56
Character interpretation in paths 57
The home directory 57
Volume and drive names 58
Mac OS volumes 58
Windows drives 58
Aliases 59
Portability issues 59
Unicode I/O 59
File Error Handling 60

5 Using ScriptUI 61

Overview 61
ScriptUI Programming Model 61
Creating a window 61
Container elements 62
Window layout 62
Adding elements to containers 63
Creation properties 64
Accessing child elements 64
Removing elements 65
Types of controls 65
Containers 65
Panel 65
Group 65
User interface controls 65
StaticText 66
EditText 66
Button 66
IconButton 66
Image 67
Checkbox 67
RadioButton 67
Progressbar 67
Slider 67
Scrollbar 68

ListBox....68
DropDownList....68
ListItem....68
Displaying icons....68
Prompts and alerts....69
Modal dialogs....69
Creating and using modal dialogs....69
Dismissing a modal dialog....70
Default and cancel elements....70
Resource Specifications....71
Defining Behavior for Controls with Event Callbacks....73
Defining event handler functions....73
Simulating user events....73
Automatic Layout....75
Default layout behavior....75
Automatic layout properties....76
Container orientation....76
Aligning children....76
Setting margins....78
Spacing between children....79
Determining a preferred size....79
Creating more complex arrangements....79
Creating dynamic content....81
Custom layout manager example....82
The AutoLayoutManager algorithm....84
Automatic layout restrictions....85
Example scripts....86
Alert box builder....86
Resource specification example....89
Localization in ScriptUI Objects....91
Variable values in localized strings....91
Enabling automatic localization....91
6 Bridge DOM Object Reference....93
App Object....94
App object properties....94
displayDialogs....94
document....94
documents....94
eventHandlers....94
favorites....94
language....94
locale....95
name....95
preferences....95
version....95
App object functions....95
beep....95
browseTo....95
buildFolderCache....96
cancelTask....96

hide 96
preflightFiles 96
purgeAllCaches 96
purgeFolderCache 96
quit 97
registerBrowseScheme 97
scheduleTask 97
system 97
Dialog Object 98
Dialog object constructor 98
Dialog object properties 98
active 98
closing 98
height 98
modal 98
title 98
width 98
Dialog object functions 98
center 98
close 98
execJS 99
open 99
place 99
print 99
Document Object 100
Document object properties 100
allowDrags 100
contentPaneMode 100
context 100
id 100
jsFuncs 100
maximized 100
minimized 100
navbars 101
noItems 101
owner 101
previewLooping 101
selections 101
showThumbnailName 101
sorts 102
status 102
thumbnail 102
thumbnailViewMode 102
visible 102
visibleThumbnails 103
visitUrl 103
Document object functions 103
bringToFront 103
close 103
deselect 104
deselectAll 104

execJS 104
maximize 104
minimize 104
refresh 104
resetToDefaultWorkspace 104
restore 104
reveal 105
select 105
selectAll 105
Event Object 106
Event object properties 106
appPath 106
document 106
favorites 106
isContext 106
location 106
object 107
type 107
url 107
where 107
Event Object Types 107
App events 107
close 107
destroy 107
Document events 108
complete 108
create 108
deselect 108
destroy 108
empty 108
failed 108
loaded 108
loading 108
open 108
select 108
stopped 108
uploading 108
Thumbnail events 109
add 109
deselect 109
hover 109
modify 109
move 109
open 109
openWith 109
preview 109
remove 110
select 110
PreferencesDialog events 110
cancel 110
create 110

destroy 110
ok 110
Favorites Object 111
Favorites object properties 111
length 111
section 111
Favorites object functions 111
addChild 111
clearAll 111
insert 111
remove 112
Metadata Object 113
Metadata object properties 113
Label 113
namespace 114
xmpPropertyName 114
Examples 114
Metadata object functions 115
applyMetadataTemplate 115
NavBar Object 116
NavBar object properties 116
file 116
height 116
jsFuncs 116
type 117
visible 117
NavBar object functions 117
add 117
execJS 118
print 118
Preferences Object 119
Preferences object properties 119
extraMetadata 119
showName 119
BackgroundColor 120
FileSize 120
HideEmptyFields 120
Label1 120
Label2 120
Label3 120
Label4 120
Label5 120
Language 120
MRUCount 120
ShowLabels 120
ShowName 120
UseLocalCaches 121
anyPropertyName 121
Preferences object functions 121
clear 121
PreferencesDialog Object 122

PreferencesDialog object functions 122
addPanel 122
close 122
Thumbnail Object 123
Thumbnail object constructor 123
Node specifiers 123
Multiple references to the same node 124
Thumbnail object properties 125
aliasType 125
children 125
container 125
creationDate 125
displayMode 125
displayPath 125
hidden 125
lastModifiedDate 125
location 126
metadata 126
mimeType 126
name 126
parent 126
path 126
spec 126
synchronousMetadata 126
type 126
Thumbnail object functions 126
copyTo 126
moveTo 127
open 127
openWith 127
refresh 127
remove 128
resolve 128
7 File and Folder Object Reference 129
Overview 129
File Object 129
File object constructors 129
File class properties 130
fs 130
File class functions 130
decode 130
encode 130
isEncodingAvailable 130
openDialog 131
saveDialog 131
File object properties 132
absoluteURI 132
alias 132
created 132
creator 132

encoding 132
eof 132
error 132
exists 132
fsName 132
hidden 132
length 132
lineFeed 132
modified 132
name 132
parent 132
path 133
readonly 133
relativeURI 133
type 133
File object functions 133
close 133
copy 133
createAlias 133
execute 133
getRelativeURI 133
open 134
openDlg 135
read 135
readch 135
readln 135
remove 135
rename 136
resolve 136
saveDlg 136
seek 136
tell 136
write 137
writeln 137
Folder Object 138
Folder object constructors 138
Folder class properties 138
appData 138
commonFiles 138
current 138
fs 138
myDocuments 138
startup 138
system 139
temp 139
trash 139
userData 139
Folder class functions 139
decode 139
encode 139
isEncodingAvailable 139

selectDialog 140
Folder object properties 140
absoluteURI 140
alias 140
created 140
error 140
exists 140
fsName 140
modified 140
name 140
parent 140
path 140
relativeURI 140
Folder object functions 141
create 141
execute 141
getFiles 141
getRelativeURI 141
remove 141
rename 141
resolve 141
selectDlg 142
File and Folder Error Messages 143
File and Folder Supported Encoding Names 144
Additional encodings 144
8 ScriptUI Object Reference 146
Overview 146
Window Class 147
Window class properties 147
coreVersion 147
version 147
Window class functions 147
alert 147
confirm 147
find 147
getResourceText 148
prompt 148
Window Object 148
Window object constructor 148
Window object properties 149
defaultElement 149
cancelElement 149
frameBounds 149
frameLocation 149
frameSize 149
Container properties 150
alignChildren 150
children 150
layout 150
margins 150

orientation ... 151
spacing ... 151
Window object functions ... 151
add ... 151
center ... 151
close ... 151
hide ... 152
notify ... 152
remove ... 152
show ... 152
Window event-handling callbacks ... 152
onClose ... 152
onMove ... 152
onMoving ... 152
onResize ... 152
onResizing ... 153
onShow ... 153
Control Objects ... 154
Control object constructors ... 154
add ... 154
Control types and creation parameters ... 154
button ... 154
checkbox ... 154
dropdownlist ... 155
edittext ... 155
group ... 156
iconbutton ... 156
image ... 156
item ... 156
listbox ... 157
panel ... 157
progressbar ... 157
radiobutton ... 158
scrollbar ... 158
slider ... 159
statictext ... 159
Control object properties ... 160
active ... 160
alignment ... 160
bounds ... 160
enabled ... 160
helpTip ... 160
icon ... 161
index ... 161
items ... 161
itemSize ... 161
jumpdelta ... 161
justify ... 161
location ... 161
maxvalue ... 161
minvalue ... 162

parent ... 162
preferredSize ... 162
properties ... 162
selected ... 162
selection ... 162
size ... 162
stepdelta ... 162
text ... 163
textselection ... 163
type ... 163
value ... 163
value ... 163
visible ... 163
Control object functions ... 164
add ... 164
find ... 164
hide ... 164
notify ... 164
remove ... 164
removeAll ... 164
show ... 164
toString ... 165
valueOf ... 165
Control event-handling callbacks ... 165
onClick ... 165
onChange ... 165
onChanging ... 165
Size and Location Objects ... 166
Bounds ... 166
Dimension ... 166
Margins ... 167
Point ... 167
LayoutManager Object ... 168
AutoLayoutManager object constructor ... 168
AutoLayoutManager object properties ... 168
AutoLayoutManager object functions ... 168
layout ... 168
MenuElement Object ... 169
MenuElement class functions ... 169
create ... 169
find ... 170
remove ... 170
Creating new menu elements ... 170
MenuElement object properties ... 171
altDown ... 171
checked ... 171
cmdDown ... 171
ctrlDown ... 171
enabled ... 171
id ... 172
onDisplay ... 172

optionDown 172
onSelect 172
shiftDown 172
text 172
type 172
Bridge menu and command identifiers 172
Bridge menu identifiers 172
Bridge submenu and command identifiers 173

9 Interapplication Communication with Scripts 180
Cross-DOM Functions 181
Cross-DOM API Reference 181
executeScript 181
open 181
openAsNew 182
print 182
quit 182
reveal 182
Application-Specific Exported Functions 182
Communicating Through Interapplication Messages 184
Sending messages 184
Receiving messages 186
Handling unsolicited messages 186
Handling responses from the message target 187
Passing values between applications 190
Passing simple types 190
Passing complex types 191
Interapplication Message API Reference 193
BridgeTalk Class 193
BridgeTalk class properties 193
appLocale 193
appName 193
appVersion 193
onReceived 194
BridgeTalk class functions 194
bringToFront 194
getSpecifier 194
getTargets 195
isRunning 196
launch 196
pump 196
BridgeTalk Message Object 197
BridgeTalk message object constructor 197
BridgeTalk message object properties 197
body 197
headers 197
sender 197
target 198
timeout 198
type 198
BridgeTalk message object callbacks 198

onError ... 198
onReceived ... 199
onResult ... 199
BridgeTalk message object functions ... 200
send ... 200
sendResult ... 200
Messaging Error Codes ... 201
Sample Workflow Automation Scripts ... 202
10 ExtendScript Tools and Features ... 203
The ExtendScript Toolkit ... 203
Configuring the Toolkit window ... 204
Selecting a debugging target ... 205
Selecting scripts ... 206
Tracking data ... 206
The JavaScript console ... 207
The call stack ... 208
The Script Editor ... 209
Mouse navigation and selection ... 209
Keyboard navigation and selection ... 209
Syntax checking ... 210
Debugging in the Toolkit ... 210
Evaluation in help tips ... 210
Controlling code execution ... 210
Visual indication of execution states ... 211
Setting breakpoints ... 212
Profiling ... 213
Dollar ($) Object ... 215
Dollar ($) object properties ... 215
build ... 215
buildDate ... 215
error ... 215
flags ... 215
global ... 215
level ... 215
locale ... 215
localize ... 216
memCache ... 216
objects ... 216
os ... 216
screens ... 216
strict ... 216
version ... 216
Dollar ($) object functions ... 216
about ... 216
bp ... 216
clearbp ... 216
gc ... 216
getenv ... 216
list ... 216
setbp ... 217

sleep ... 217
summary ... 217
write ... 217
writeln ... 217
Object statistics ... 217
ExtendScript Reflection Interface ... 219
Reflection Object ... 219
Reflection object properties ... 219
description ... 219
help ... 219
methods ... 219
name ... 219
properties ... 219
Reflection object functions ... 219
find ... 219
ReflectionInfo Object ... 220
ReflectionInfo object properties ... 220
arguments ... 220
dataType ... 220
defaultValue ... 220
description ... 220
help ... 221
isCollection ... 221
max ... 221
min ... 221
name ... 221
type ... 221
Localizing ExtendScript Strings ... 222
Variable values in localized strings ... 222
Enabling automatic localization ... 222
Locale names ... 223
Testing localization ... 224
Global localize function ... 225
localize ... 225
User Notification Helper Functions ... 226
Global alert function ... 226
alert ... 226
Global confirm function ... 227
confirm ... 227
Global prompt function ... 227
prompt ... 227
Specifying Measurement Values ... 229
UnitValue Object ... 229
UnitValue object constructor ... 229
UnitValue object properties ... 230
baseUnit ... 230
type ... 230
value ... 230
UnitValue object functions ... 230
as ... 230
convert ... 230

Converting pixel and percentage values ... 230
Computing with unit values ... 231
Modular Programming Support ... 233
Preprocessor directives ... 233
#engine *name* ... 233
#include *file* ... 233
#includepath *path* ... 234
#script *name* ... 234
#strict on ... 234
#target *name* ... 234
Importing and exporting between scripts ... 234
Operator Overloading ... 236
Application and Namespace Specifiers ... 237
Application specifiers ... 237
Namespace specifiers ... 238
Script Locations and Checking Application Installation ... 238
Index ... 240

Welcome

Welcome to the *Bridge JavaScript Reference*. This book describes how to use JavaScript to manipulate and extend Adobe® Bridge for Adobe Creative Suite 2.

About This Book

The *Bridge JavaScript Reference* describes how to use the scripting API to extend and manipulate Adobe Bridge, but it is not a user's guide for the Bridge application and its user interface.

This book provides complete reference information for the JavaScript objects, properties, and functions defined by Adobe Bridge, and for various utilities and tools that are part of ExtendScript, the Adobe extended implementation of JavaScript.

This book also describes how to use the interapplication communication framework that is defined by Adobe Bridge and included in each Adobe Creative Suite 2 application. You can use this framework to write scripts that call on functionality from different applications, or to send scripts and data from one application to another. A set of Sample Workflow Automation Scripts is provided with Adobe Creative Suite 2, which demonstrate how scripts can be used to create a workflow that takes advantage of functionality in different applications.

Who should read this book

This book is for developers who want to extend the capabilities of Adobe Bridge using JavaScript, call Bridge functionality from scripts, and use scripts to communicate between Adobe Creative Suite 2 applications. It assumes a general familiarity with the following:

- JavaScript
- C and C++ programming
- Adobe Bridge
- Any other Adobe Creative Suite 2 applications you are using, such as Illustrator® CS2, Photoshop® CS2, or InDesign® CS2. The scripting API details for each application are included with the scripting documentation for that product.

What is in this book

This book provides conceptual information about the scripting Adobe Bridge and detailed reference information about the JavaScript objects that Adobe Bridge provides. It also provides both usage and reference information for the tools, utilities, and objects that are part of ExtendScript, the Adobe extended implementation of JavaScript.

This book contains the following chapters:

- Chapter 1, "Scripting Bridge," introduces some important concepts in Adobe Bridge scripting and describes the Bridge JavaScript document object model (DOM).
- Chapter 2, "Event Handling and Script-Defined Browse Schemes," describes how Adobe Bridge generates user-interaction events, and how you can respond to these events by defining handlers in

your scripts. In addition, it describes how to define browse schemes that allow you to extend or modify what is shown in the Bridge Favorites pane.

- Chapter 3, "Creating a User Interface," describes the various options available to scripts for interaction with Bridge users, such as dialog boxes and navigation bars.
- Chapter 4, "Using File and Folder Objects," describes how to use the ExtendScript objects that provide platform-independent access to the underlying file system.
- Chapter 5, "Using ScriptUI," describes how to use the ExtendScript user interface module, a set of objects which provide windows and user-interface controls for the scripting environment.
- Chapter 6, "Bridge DOM Object Reference," provides a complete API reference for the objects, properties, and functions defined in the Bridge document object model.
- Chapter 7, "File and Folder Object Reference," provides a complete API reference for the ExtendScript file-system access objects, properties, and functions.
- Chapter 8, "ScriptUI Object Reference," provides a complete API reference for the ExtendScript user-interface objects, properties, and functions.
- Chapter 9, "Interapplication Communication with Scripts," describes how to use the interapplication communication framework, and provides a complete API reference for the Cross-DOM and for the messaging framework.
- Chapter 10, "ExtendScript Tools and Features," describes the ExtendScript Toolkit debugging environment, and provides a complete API reference for the ExtendScript utilities and features that are available to all Adobe Creative Suite 2 applications.

Document conventions

Typographical conventions

`Monospaced font`	Literal values and code, such as JavaScript code, HTML code, filenames, and pathnames.
Italics	Variables or placeholders in code. For example, in `name="`*`myName`*`"`, the text *`myName`* represents a value you are expected to supply, such as `name="Fred"`. Also indicates the first occurrence of a new term.
Blue underlined text	A hyperlink you can click to go to a related section in this book or to a URL in your web browser.
Sans-serif bold font	The names of Bridge UI elements (menus, menu items, and buttons). The > symbol is used as shorthand notation for navigating to menu items. For example, **Edit > Cut** refers to the **Cut** item in the **Edit** menu.

Note: Notes highlight important points that deserve extra attention.

JavaScript conventions

This reference does not list properties and methods provided by the JavaScript language itself. For example, it is common for JavaScript objects to provide a `toString` method, and many of the objects the SDK supplies implement this method. However, this book does not describe such methods unless they differ from the standard JavaScript implementation.

Similarly, because most objects provided by the SDK have a `name` property, the reference does not list `name` properties explicitly.

When a JavaScript function returns a value, it is listed.When there is no return value listed, the function does not return a value.

Where to go for more information

This book does not describe the JavaScript language. For documentation of the JavaScript language or descriptions of how to use it, see any of numerous works on this subject, including the following:

JavaScript: The Definitive Guide, 4th Edition; Flanagan, D.; O'Reilly 2001; ISBN 0-596-00048-0

JavaScript Programmer's Reference; Wootton, C.; Wrox 2001; ISBN 1-861004-59-1

JavaScript Bible. *5th Edition*; Goodman, D. and Morrison, M.; John Wiley and Sons1998; ISBN 0-7645-57432

1 Scripting Bridge

This chapter introduces some important concepts in Adobe Bridge scripting and describes the Bridge JavaScript document object model (DOM).

Scripting Overview

Adobe Bridge provides a configurable, extensible browser platform that allows users to search for and select files by navigating among files and folders in the local file system, those on remote file systems, and also web pages accessible over the Internet.

Bridge is integrated with Adobe Creative Suite 2, and various applications bring up the Bridge browser window in response to specific user actions that require file selection. You can also bring up a Bridge browser window independently, by invoking it interactively or through a script.

The Bridge browser is highly configurable and extensible, using ExtendScript, the Adobe extended implementation of JavaScript. ExtendScript files are distinguished by the `.jsx` extension. ExtendScript offers all standard JavaScript features, plus additional features and utilities, such as:

- Platform-independent file and folder representation
- Tools for building a user interface to a script
- An interactive development and debugging environment (the ExtendScript Toolkit)

You can use ExtendScript to manipulate browser windows and their contents programmatically, and to change and extend their functionality. This manual describes what you can do, and provides a complete reference for the ExtendScript objects and functions that you can use to program Bridge.

Executing Scripts

Bridge executes scripts in any of these ways:

- On startup, Bridge executes all JSX files that it finds in the startup folders.
  - In Windows®, the startup folders are:

    ```
    %APPDATA%\Adobe\StartupScripts
    %APPDATA%\Adobe\StartupScripts\bridge\version
    ```

  - In Mac OS®, the startup folders are:

    ```
    ~/Library/Application Support/Adobe/StartupScripts/
    ~/Library/Application Support/Adobe/StartupScripts/bridge/version/
    ```

  Note: If your script is in the main startup folder, it is also executed by all other Adobe Creative Suite 2 applications at startup; see Script Locations and Checking Application Installation.

  The *version* portion of the Bridge-specific folder path is an exact version number. That is, scripts in the folder `bridge/1.5` are executed only by Bridge version 1.5, and so on.

- You can pass a script to the Bridge executable to be executed on startup, by dragging the JSX file icon onto the Bridge executable file icon or shortcut. This script is executed after all startup scripts.

- When the Bridge browser window displays a JSX file, you can double-click that file thumbnail to run the script in its target application. It runs in Bridge if the script specifies Bridge as its target application by including the directive:

  ```
  #target "bridge"
  ```

  If the script specifies another Adobe Creative Suite 2 application as its target, ExtendScript starts that application if necessary. If the script does not specify a target application, it opens in the ExtendScript Toolkit. For details, see Preprocessor directives and The ExtendScript Toolkit.
- You can load and run a script in the ExtendScript Toolkit, specifying Bridge as the target application. For details, see The ExtendScript Toolkit.
- You can add a menu command that runs a script to a menu or submenu in the Bridge browser, using the MenuElement Object.

Communicating with Other Applications

Adobe Bridge provides an interapplication communication framework, a way for scripts to communicate with other Adobe applications, from Bridge or among themselves.

- A script can call certain basic functions exported by all Adobe Creative Suite 2 applications. For example, a Bridge script could ask the user to select an image file, then open that file in Photoshop® or Illustrator® by calling the `photoshop.open` or `illustrator.open` function. These basic exported functions are called the *Cross DOM*.
- Individual applications export additional functions to make more complex functionality available to scripts. For example, a Bridge script can request a photo-merge operation in Photoshop by calling `photoshop.photomerge` with a set of selected image files. The set of functions available for each application varies widely.
- A messaging protocol provides a general and extensible framework for passing any kind of data between *messaging enabled* applications. All Creative Suite 2 applications are messaging enabled. You can send messages that contain JavaScript scripts. The target application can evaluate a script that it receives, and send results back in a response message.

For additional information, see Chapter 9, “Interapplication Communication with Scripts."

The Bridge Document Object Model

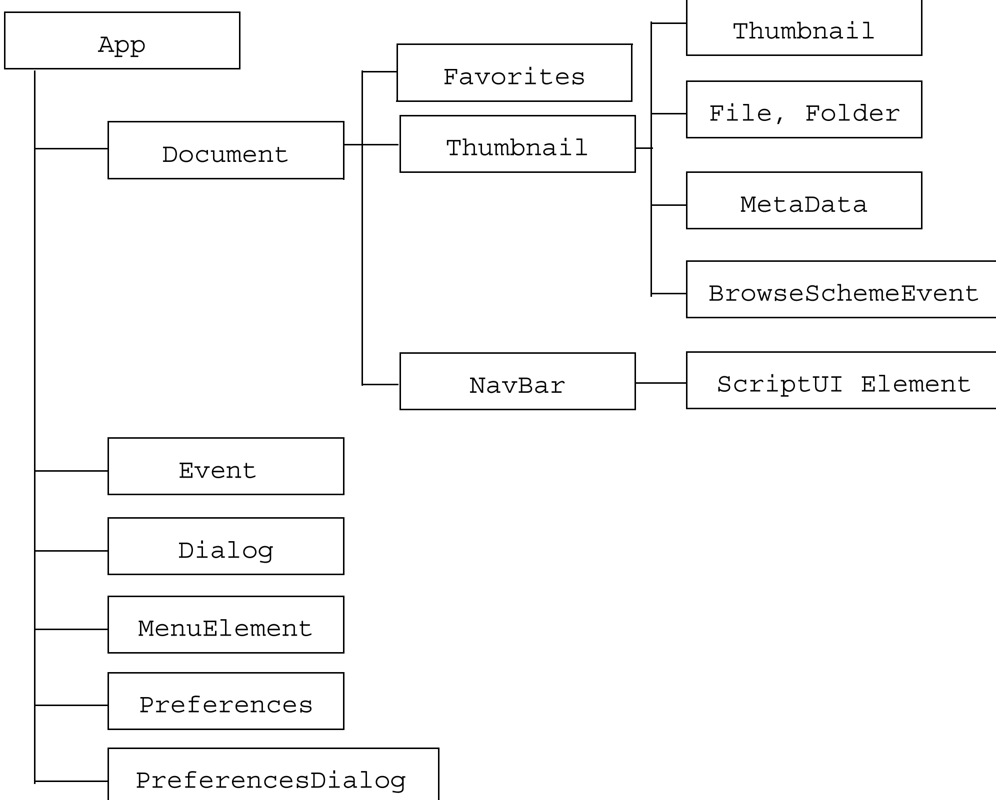
A document object model (DOM) is an application programming interface (API), which allows you to programmatically access various components of a *document* (as defined for that application) through a scripting language such as JavaScript.

Each application in the Creative Suite 2 has its own DOM, which consists of a hierarchical representation of the application, and of the documents used in the application. The DOM allows you to programmatically access and manipulate the document and its components. Since the use of a document varies for each application, the DOM terminology varies for each application. For example, each application's DOM includes a `Document` class, but the object referred to is different for each application, and the `Document` class has different properties and methods for each application.

Applications typically define a `Document` class to deal with files of a particular type, such as HTML pages, images, or PDF documents. However, Bridge uses a different approach. In the Bridge DOM, the `Document` class refers to a Bridge browser window, and the properties and methods of `Document` refer to various components of the Bridge user interface (UI). The browser window displays icons that reference the files

that other applications consider documents—HTML pages, images, PDFs, and so on. In the Bridge DOM, these icons are represented by the `Thumbnail` class.

Having a good understanding of the Bridge DOM, and how each aspect of the API relates to the Bridge browser, will greatly enhance your ability to write scripts. In the Bridge DOM shown below, each node in the hierarchy represents a class in the Bridge DOM API.



The application and documents

The Bridge App Object is the root of the hierarchy, and represents the Bridge application. A single global instance of the `App` class, named `app`, is created when the application is started, and provides access to global values. Even though the user can create multiple Bridge browser windows by selecting the **File** > **New Window** command, making it appear that separate Bridge applications are running in parallel, only a single instance of the application is running, which is reflected by a single instance of the `app` object.

The Document Object represents a Bridge browser window. Each time a user selects **File** > **New Window**, a new `document` object is created. When multiple Bridge browser windows are open, the user can select which window to use by clicking the window to make it active. In a script, you can access the active, or most recently used, browser window through the `app.document` property. The set of all open browser windows is available through the `app.documents` array.

Thumbnails in documents

The Thumbnail Object type represents a node in the browser navigation hierarchy. It typically represents a a file or folder, but can also be associated with a web page. A document contains various collections of `Thumbnail` objects.

`Thumbnail` objects can contain other `Thumbnail` objects, as for example, when a folder contains files. In this case, the `children` property of the `Thumbnail` contains a hierarchy of `Thumbnail` objects that represent files and folders contained in the folder.

The Folders pane shows the full navigation hierarchy of folders and subfolders for the local file system. A script cannot add nodes to the Folders pane (except by creating new folders on disk), but your script can access the selected thumbnail through the `app.document.thumbnail` property. It can walk the navigation hierarchy by accessing the `parent` and `children` properties of each `Thumbnail` object.

The Favorites pane shows a selection of high-level nodes, some predefined and some chosen by the user. These can represent web pages and remote folders, as well as local folders. The Favorites Object represents the navigation nodes in the Favorites pane. A document contains a single `Favorites` object, which is an array of `Thumbnail` objects. Access the `Favorites` object through `app.favorites`.

A script can add thumbnails to the lower part of the Favorites pane by using the `Favorites` object's `insert` method, and one level of sub-nodes using the `addChild` method. A subnode can be any thumbnail; it does not have to be part of the root node's `children` hierarchy.

Thumbnails as node references

The Thumbnail Object represents a navigation node. Thumbnails can represent entities such as files and folders, accessed through a local or remote file system, or web pages accessed over the internet and displayed in an embedded web browser. Each `Thumbnail` object is associated with one of the following types of node identifier, which determines what happens when the user selects the icon:

- `File` or `folder` objects; see Chapter 4, "Using File and Folder Objects," and Chapter 7, "File and Folder Object Reference." Clicking a folder thumbnail in the Folders or Favorites pane displays the contents of the folder in the Content pane.
- URLs. Clicking the thumbnail in the Folders or Favorites pane displays the web page associated with the URL in the Content pane. This can be a local or remote HTML page. See Displaying HTML in the Content Pane.
- Version Cue® nodes. Clicking a thumbnail in the Folders pane displays the contents of the Version Cue workspace in the Content pane.
- Script-defined navigation nodes, which are associated with script-defined browse schemes. The display in the Content pane is determined by the browse scheme's associated handler. See Script-Defined Browse Schemes.

Using and accessing thumbnails

Thumbnails are used in a number of ways within a browser window, and the objects are referenced according to their use. For example:

- Access thumbnails that appear in the Favorites pane through `app.favorites`
- Access a thumbnail that is selected in the Folders pane through `app.document.thumbnail`
- Access thumbnails that appear in the Content pane through `app.document.thumbnail.children`

- Access thumbnails that have been selected in the Content pane through `app.document.selected`
- Access thumbnails that are associated with a context menu through `app.document.context`

A `Thumbnail` object is associated with a Metadata Object, which allows you to access the external data associated with the associated file, such a copyright owner, author, or camera settings. The `metadata` object also allows access to an image thumbnail's label string, which you can define and set through the Bridge browser.

User interface objects

Your script can display information to or collect information from the user by configuring the supplied navigation bars, or by creating and displaying dialogs.

ExtendScript provides a set of user-interface objects in the ScriptUI module, which defines windows and user-interface controls. You can use these objects to define a user interface for your application, in the form of popup dialogs, persistent dialogs (called palettes), or as part of navigation bars. The usage of the ScriptUI objects is discussed in Chapter 5, "Using ScriptUI," and complete syntax details are provided in Chapter 8, "ScriptUI Object Reference."

You can also define a user interface using standard HTML. When integrating your user interface with the Bridge browser, you can use either ScriptUI or HTML controls for any window or pane, but cannot mix the two. For a complete discussion, see Chapter 3, "Creating a User Interface."

In addition to displaying a user interface for your script, you can script user interactions by extending the Bridge menus.

Navigation bars

The Bridge navigation bar immediately below the menubar cannot be scripted, but there are two configurable navigation bars, above and below the Content pane. They are represented by NavBar Objects, which you can access through the `Document` object's navbars property.

By default, the navigation bars are hidden and empty.

- You can show and hide a navigation bar by setting the object's `visible` property.
- You can configure a navigation bar to display either ScriptUI user-interface controls, or straight HTML controls. It cannot mix the two.
  - To display ScriptUI controls, set the `type` property to `"scriptui"`, then use the `NavBar.add` method to add controls.
  - To display HTML controls, set the `type` property to `"html"`, and the `file` property to the HTML file that defines the page you want to display.

You can program the controls to display information to or collect information from the user. For additional details, see Navigation Bars.

Dialogs

Your script can define dialogs to display information to or get information from the user. There are two ways to define these:

- **ScriptUI Dialogs**: Use the ScriptUI Window Object to define a dialog that displays ScriptUI controls.

- **Bridge Dialogs**: The Dialog Object represents a window that displays an HTML page, rather than ScriptUI controls.

You can invoke ScriptUI dialogs from a script as *modal* or *nonmodal* dialogs.

- A modal dialog retains the input focus, and does not allow the user to interact with any other application windows until the dialog is dismissed. The function that invokes it does not return until the dialog is dismissed.
- A nonmodal dialog (known in ScriptUI as a *palette*), does not keep the input focus. The user can interact with other application windows while the dialog is up. The function that invokes it returns immediately, leaving the dialog on screen until the user or script closes it.

For details of programming dialogs, see Chapter 5, "Using ScriptUI," and Displaying HTML in Bridge Dialogs.

Menus

The MenuElement Object allows you to add new menus and commands. A script cannot remove or alter the behavior of predefined menu items, but you can add, remove, and modify script-defined menus and commands.

Most menubar menus and context menus can be extended by creating new `MenuElement` objects that reference existing menus and menu items. The identifiers of all menus and menu items that are accessible to scripts are listed with the description of the `MenuElement` object.

Events

When interacting with Bridge, a user takes actions such as copying a file, or creating a new Bridge browser window. For most of these actions, Bridge triggers a *user-interaction event*, represented by an Event Object of a particular *event type*, with a particular *target object*, such as an `App`, `Document`, or `Thumbnail` object. Some function calls can also trigger events.

Bridge defines default behavior for user-interaction events. You can extend or override the default behavior by defining and registering an *event-handler function*. This function receives the `Event` object as an argument, and returns a value that tells Bridge whether to continue with the default behavior or to ignore it.

For more information on event handling, see Event Handling in Bridge.

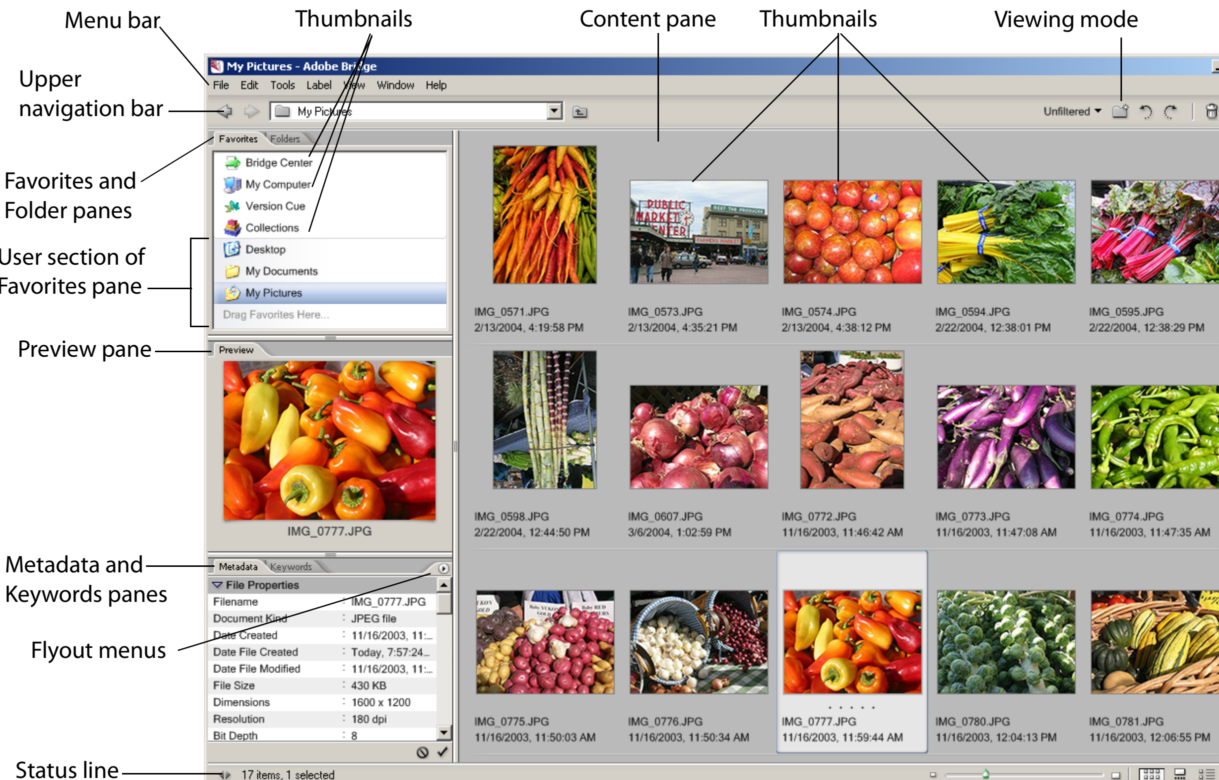
Application preferences

The Preferences Object allows a script to access Bridge application preferences. These are the values that can be viewed interactively in the Preferences dialog, in response to the **Edit** > **Preferences** command. The settings are stored and persist across sessions. Your script can use the `Preferences` object to view or set existing preferences, or to add new preference fields. In general, when you modify persistent preference values, the new settings are not reflected in the browser window until the Bridge application is restarted.

When the user brings up a Preferences dialog, Bridge invokes a ScriptUI dialog window, and generates a `create` event with the PreferencesDialog Object as its target. You can define and register an event handler for this event that uses the object's `add` method to add a ScriptUI panel containing ScriptUI controls that show and allow the user to modify script-defined preference values.

The Bridge DOM and the Bridge Browser Window

The following figure identifies parts of the Bridge browser window in a default configuration.



The following table describes how the Bridge API maps to the various parts and features of the Bridge browser window, and how a script can access each part or feature through the Bridge DOM.

Bridge window area	Purpose	Bridge DOM control
Browser window	Displays files, folders, and web pages.	Represented by the `Document` object. Current browser window is in `app.document`.
Favorites pane	Provides a place for users to drag and drop favorite items (in the bottom half of the pane). Displays only top-level containers and one level of subnodes.	Access the visible thumbnails through the `app.favorites` property, and traverse the hierarchy through the `Thumbnail.children` properties. The `Favorites` object allows you to add thumbnails to this pane. User interaction with a thumbnail in this pane generates an `event` with a `Thumbnail` target object and a location of `favorites`. You can define your own *browse scheme* for a thumbnail in this pane, which allows you to extend or redefine the thumbnail's behavior. See Script-Defined Browse Schemes.

Folders pane	Displays the navigation hierarchy and controls navigation. Displays only containers (such as folders and subfolders).	Access the currently selected thumbnail in the Folder pane through the `Document.thumbnail` property. Traverse the hierarchy through the `Thumbnail.parent` and `Thumbnail.children` properties. User interaction with a thumbnail in this pane generates an `event` with a `Thumbnail` target object and a location of `document`.
Preview pane	Displays image previews.	User interaction with a thumbnail in this pane generates an `event` with a `Thumbnail` target object and a location of `preview`.
Metadata pane	Displays metadata information.	Metadata is displayed and written for a specific thumbnail. Access metadata from the `Thumbnail.metadata` property.
Keywords pane	Displays keyword information.	Not accessible to scripts.
Content pane	Displays navigation results when you select a node in the Folders or Favorites pane, or when you click on a navigable node (such as a folder) in the Content pane itself. Displays both containers (such as subfolders) and leaf nodes (such as files and images). Can display HTML pages for web-type thumbnails.	Controlled by the browse scheme associated with the thumbnail selected in the Folders or Favorites pane. The browse scheme uses the `Thumbnail` properties `displayMode` and `displayPath` to determine what appears the Content pane, and how it appears. ● When the selected thumbnail has `displayMode="filesystem"`, its children appear as icons in the Content pane. When a folder is selected in the Folders pane, access the current contents of the Content pane through `app.document.thumbnail.children[]`. ● When the selected thumbnail has `displayMode="web"`, the associated HTML page (local or remote) appears in the Content pane. User interaction with a thumbnail in this pane generates an `event` with a `Thumbnail` target object and a location of `document`. The selected thumbnails are available through `Document.selections`.
Thumbnails	The labelled icons that appear in the Folders/Favorites panes as navigation nodes, and in the Content pane to represent files and folders.	The `Thumbnail` object represents a node in the browser navigation hierarchy. Thumbnails can represent `File` or `Folder` objects, Version Cue nodes, URLs, or script-defined nodes associated with script-defined browse schemes.

Menubar	The menubar at the top of the Bridge browser window.	While this is not an object under direct `Document` control, you can add menus and commands using the `MenuElement` object, by referring to existing menus and commands.
Context menus	The right-click menus associated with thumbnails, and flyout menus for some tab panels.	You can add submenus and commands to these menus using the `MenuElement` object, by referring to existing commands.
Browser window upper navigation bar	The navigation bar immediately under the menubar. Not configurable.	Not accessible to scripts.
Status line	The bottom bar on the Bridge browser window.	Controlled by `Document.status`.
Content Pane Viewing Control	The bar next to the status line, where the viewing mode of the Content pane is controlled.	The view mode is controlled by `app.document.thumbnailViewMode`.
Top and bottom navigation bar s	Two configurable navigation bars that can appear above and below the Content pane.	Represented by predefined `NavBar` objects, accessed through the `Document.navbars` property. By default, the navigation bars are invisible. You can make a bar visible, and add ScriptUI or HTML UI controls to it.

2 Event Handling and Script-Defined Browse Schemes

A script can extend and modify the behavior of Adobe Bridge at a number of levels:

- At the highest level, the basic behavior of a thumbnail icon is determined by its browse scheme. There are several predefined browse schemes that provide the most common kinds of behavior, such as displaying the contents of a folder, or displaying a web page (see Thumbnails as node references). You can select one of the predefined browse schemes to create a thumbnail that has the basic default behavior you require. (see Thumbnail object constructor).
- You can intercept user-interaction events to extend or override the default behavior. These events include actions on thumbnails (such as selecting them), and also actions on the application (quitting) and on the browser window (such as activating it in the windowing system). See Event Handling in Bridge below.
- You can modify the displayed thumbnails in the Favorites pane by defining thumbnails with Script-Defined Browse Schemes.

Event Handling in Bridge

When a user takes certain actions in Bridge, such as copying a file, or creating a new Bridge browser window, Bridge generates a user-interaction event. You can modify the way Bridge responds to these events by defining your own event handlers. Scripts can also generate events through function calls, that simulate user activity, such as the Thumbnail Object's open method, or the Document Object's select method.

Defining event handlers

An event-handler function takes one argument, an Event Object. This object, which is passed to your registered handler when the event occurs, contains all of the context information about the event, such as which type of event occurred, the target object that generated it, and where that object was located within the browser window.

Your handler returns an object with a boolean `handled` property.

- When an event handler returns `{handled:true}`, Bridge does not look for any more handlers, nor does it execute the default handler.
- When an event handler returns `{handled:false}`, Bridge continues to look for registered handlers, and if no more script-defined handlers are registered, it executes the default handler. This is the default behavior if your handler does not return a value.

Using this mechanism, you can extend the default behavior of the Bridge objects. For example, when the user quits the Bridge application, your `destroy` event handler can take additional actions, such as cleaning up structures you have made, or displaying status information. To extend the default behavior, your handler returns the object `{handled:false}`.

In many cases, such as Thumbnail events, you can use the event handler to override the default behavior. You do this by returning the object `{handled:true}`, which prevents Bridge from executing the default handler.

For some events, such as `Document` events, you cannot override the default behavior of the event. Even if your handler returns `{handled:true}`, the default behavior still executes when your handler has finished. The `{handled:true}` return value does, however, prevent Bridge from executing any subsequent script-registered event handlers.

Registering event handlers

To register an event-handler function you have defined, create an `EventHandler` object and add it to the array `app.eventHandlers`. An `EventHandler` is a simple JavaScript object with a `handler` property that specifies the name of the event-handler function. There is no constructor, it is a simple script-defined object. For example:

```
var myEventHandler = { handler: doThisEvent };
app.eventHandlers.push (myEventHandler);
```

You can write one handler that responds to many different events, or write one handler for each type of event. When an event occurs, Bridge iterates through the `app.eventHandlers` array, trying each handler in sequence, passing in the triggering event object. If one of the event handlers returns `{handled:true}` Bridge stops the iteration.

Event handling examples

➤ Example: Separate handlers for separate events

This script defines separate event handlers for two `Document` events, `select` and `create`, and registers them by adding them to the `app.eventHandlers` array. These handlers return `{handled:true}`, which in this case does not override the default behavior of the events, but does tell the event handling mechanism to stop processing subsequent script-defined event handlers.

```
// define handlers
onCreateDocument = function( event ) {
  if( event.object.constructor.name == "Document" ) {
    if( event.type == "create" ) {
      // Action to take in the event of Document create
      Window.alert("new document");
      return {handled:true}; //stop processing event handlers
    }
  }
}

onFocusDocument = function( event ) {
  if( event.object.constructor.name == "Document" ) {
    if( event.type == "select" ) {
      // Action to take in the event of Document focus
      app.beep();
      return {handled:true};//stop processing event handlers
    }
  }
}
// register handlers
app.eventHandlers.push( { handler: onCreateDocument} );
app.eventHandlers.push( { handler: onFocusDocument} );
```

➤ Example: One handler for all document events

This script defines one event handler to handle several `Document` events, distinguishing the events internally.

```
onDocumentEvent = function( event ){
   if( event.object.constructor.name == "Document" ){
      if( event.type == "create" ) {
         // Action to take in the event of Document create
         Window.alert("new document");
         return {handled:true};//stop processing event handlers
      }
      else if( event.type == "focus" ){
         // Action to take in the event of Document focus
         app.beep();
         return {handled:true};//stop processing event handlers
      }
      else if( event.type == "select" ){
      // Action to take in the event of Document select
      Window.alert("document was selected");
      return {handled:true};//stop processing event handlers
      }
   }
}
// register the handler
allDocEventHandlers = { handler: onDocumentEvent };
app.eventHandlers.push( allDocEventHandlers );
```

➤ Example: Cancelling a quit operation using app close

This script defines a handler for the `App close` event, which occurs when Bridge receives a request for application shutdown. In this case, if the result object contains `handled: true`, the shutdown is cancelled. This handler queries the user, and only continues with the operation if the user confirms.

```
var myHandler = function(event){
   if ((event.type == 'close') && (event.object instanceof App)) {
      return { handled: Window.confirm("Really quit?") };
   }
   return { handled: false };
}
app.eventHandlers.push( { handler: myHandler } );
```

➤ Example: Adding a service to Bridge using document create

This script defines a handler for the `Document create` event that adds a navigation bar to all new browser windows. The script is placed in the startup script directory, so that it runs each time Bridge is invoked. Whenever the user creates a new browser window, the event handler adds the navigation bar to it.

- The navigation bar uses ScriptUI objects to display a text field in which the user can type a path. When the user clicks **Go** or **New** in the navigation bar, the button's onClick handler navigates to that path.
- A handler for the `Thumbnail select` event updates the navigation bar so that it always shows the path of the currently selected thumbnail.

Note: This example is a variation of the navigation bar examples given in Chapter 3, "Creating a User Interface." See that chapter for more discussion of navigation bars.

```
// Adds a path bar to all browser windows, which allows you to navigate
// by typing a path, and shows the path of the current selection

// Change current document to browse the path in the path bar
//  (called when the Go button is clicked)
function browseToPath( message ) {
   try {
      app.document.thumbnail = new Thumbnail(
         app.document.topNavbar.pathPanel.browseField.text );
   } catch ( error ) {
      Window.alert (error );
   }
}
// Open a new window to browse the path on the path bar.
//  (called when the New button is clicked)
function browseToPathNewWindow( message ) {
   try {
      app.browseTo( app.document.topNavbar.pathPanel.browseField.text );
   } catch (error ) {
      Window.alert (error);
   }
}
// Create the PathBar panel on a nav bar
function addTopBarPanel(bar) {
   bar.pathPanel = bar.add( "panel", [5,5, 830, 35], undefined);
   bar.pathPanel.browseField = bar.pathPanel.add( "edittext",
      [3, 3, 700, 22],"");
   bar.pathPanel.BrowseBtn = bar.pathPanel.add( "button",
      [710, 3, 760, 22],"Go");
   bar.pathPanel.BrowseBtnNewWin = bar.pathPanel.add( "button",
      [770, 3, 820, 22],"New");
   bar.pathPanel.BrowseBtn.onClick = browseToPath;
   bar.pathPanel.BrowseBtnNewWin.onClick = browseToPathNewWindow;
}
// Create the PathBar on the top navbar of the document
function addNavBar(doc) {
   var topbar = doc.topNavbar;
   addTopBarPanel(topbar );
   topbar.visible = true;
   topbar.pathPanel.browseField.text = theDocument.thumbnail.path;
}
// Handle  document create event to add path bar to new browser windows.
onDocumentEvent = function( event ){
   try {
      if( (event != undefined) && (event.object != undefined)
            && (event.type != undefined) ) {
         if( event.object.constructor.name == "Document" ){
            if( event.type == "create" ) {
            // Action to take on document creation
               addNavBar( event.object );
            }
         }
         // (check for no documents to avoid thumbnail event
         //  that occurs before first doc opens)
         else if( event.object.constructor.name == "Thumbnail" &&
```

```
                    app.documents.length != 0 ) {
            if( event.type == "select") {
            // Action to take on Thumbnail selection
            // update the path bar to show the current path
               if(app.document.topNavbar.pathPanel != 0 ) {
                  if(event.object.container) {
                     app.document.topNavbar.pathPanel.browseField.text =
                           event.object.path;
                  }
                  else {
                     app.document.topNavbar.pathPanel.browseField.text =
                           event.object.parent.path;
                  }
               }
            }
         }
         else {
            if( event == undefined ) {
               Window.alert ("null event");
            }
         }
      }
   } catch (error) {
      Window.alert (error);
   }
}
// Register the event handler
app.eventHandlers.push( { handler: onDocumentEvent} );
```

User-interface events

The event handling mechanism described in this chapter applies only to the Bridge DOM objects. If your script defines its own user interface, events are handled differently, depending on what kind of object generated them:

- For events generated by ScriptUI objects (such as controls in the navigation bar), see Chapter 5, "Using ScriptUI."
- For events generated by menu elements, see MenuElement Object.
- Events generated by HTML controls in HTML navigation bars or dialogs are handled by their own HTML-defined handlers. These can access Bridge DOM objects through a callback mechanism. See Displaying HTML in Bridge.

Script-Defined Browse Schemes

A browse scheme (or navigation) protocol determines how Bridge performs navigation for each thumbnail—that is, exactly what happens internally when a user opens, closes, clicks, double-clicks, or otherwise interacts with the `Thumbnail` object in the Folders, Favorites, or Content pane.

`Thumbnail` objects that represent `File` or `Folder` objects, Version Cue nodes, and URLs have predefined browse schemes. You can also create your own script-defined browse scheme, register it with the application, and specify it as part of the path when you create `Thumbnail` objects.

Creating script-defined browse schemes allows you to create a navigation hierarchy that includes different types of `Thumbnail` objects. This would typically be used to create a web service, where the root node displays an HTML page that allows the user to make choices regarding the contents of contained folders.

For example, you can create a hierarchy that contains a top-level `Thumbnail` that points to a URL, and sub-node `Thumbnail` objects that point to file system folders, which you might use to save files downloaded from the URL. You can use the Favorites Object's insert function to add the root of your script-defined thumbnail hierarchy to the Favorites pane, where users can interact with it.

Selecting a thumbnail that points to a URL displays the remote page in an embedded browser in the Content pane. Similarly, a thumbnail with a script-defined browse scheme can be used to display a locally-defined HTML page in the Content pane. For details, see User Interface Options for Scripts.

Defining and registering a browse scheme

The function `app.registerBrowseScheme` registers a new script-defined browse scheme protocol with Bridge. To create a new protocol, simply assign it a name and register that name. You can then create `Thumbnail` objects that use the registered name as the protocol part of the `path` argument (which takes the form *`browseProtocol`*`://`*`pathSpecifier`*).

For example, the following code registers a new browse scheme, `bank`, and creates two new `Thumbnail` objects that use that browse scheme.

```
app.registerBrowseScheme( "bank" );

var bankRoot = new Thumbnail( "bank://root", "My Bank" );
var transactions = new Thumbnail( "bank://transactions", "Transactions" );
```

Thumbnails with script-defined browse-scheme protocols can be added to the top level of the Favorites pane using the `Favorites.insert` method, or as subnodes of top-level nodes, if they are `Thumbnail` objects with the same script-defined browse-scheme protocol. To add a subnode, use the `Favorites.addChild` method. You cannot make a subnode from a `Thumbnail` that uses a different browse scheme from its parent node, or one that is associated with a `Folder` object.

This simple definition does not associate any new behavior with the browse scheme, but can still be useful. For this example to make use of existing browsing functionality in a new way, it simply assigns different display styles to the `Thumbnail` objects at different levels of the hierarchy.

The following code adds the new `Thumbnail` objects to the Favorites pane. The first, `bankRoot`, is added to the top level. The second, `transactions`, is added both as a thumbnail child and as a sub-node to `bankRoot`:

```
bankRoot.displayPath = "http://www.mybank.com";
bankRoot.displayMode = "web";
transactions.displayPath = "/C/BankTransactions";
```

```
transactions.displayMode = "filesystem";

bankRoot.insert(transactions); //add thumbnail child

app.favorites.insert( bankRoot );  //add node
app.favorites.addChild( bankRoot, transactions ); // add subnode
```

The top-level `thumbnail` displays a URL, and the child displays a file. Because no special handler has been associated with this browse scheme, the objects respond to events using the default browsing and event-handling behavior:

- When the user selects the `bankRoot` thumbnail, Bridge navigates to the URL `http://www.mybank.com`, and displays it in the Content pane. (You could set `displayPath` to point to a locally-defined HTML page—see Displaying HTML in the Content Pane.)
- When the user selects the `transactions` thumbnail, Bridge navigates to the directory `C:\BankTransactions`, and displays its contents in the Content pane.

3 Creating a User Interface

User Interface Options for Scripts

The Bridge scripting environment provides a number of options for interacting with users. You can handle a user's interaction with Bridge objects, such as thumbnails, through the event-handling mechanism (Event Handling in Bridge), and you can extend the Bridge menus, adding your own submenus and commands (MenuElement Object).

However, if you want to display your own window or pane to the user, you can do so in several ways: by creating popup or persistent *dialogs*; by configuring and displaying predefined *navigation bars*; or by defining user-interface controls to be displayed in the Content pane, in response to selection of specially defined thumbnails.

You can define user-interface controls in any of these places in two ways:

- **ScriptUI Elements**: ScriptUI is a module that defines windows and user-interface controls. You can create ScriptUI Dialogs Boxes and populate them with ScriptUI controls, or add ScriptUI controls to the existing Navigation Bars.

  If you use ScriptUI controls, you can take advantage of the ExtendScript localization feature; see Localization in ScriptUI Objects.
- **HTML Pages**: An HTML page can contain standard HTML user-interface controls. You can display HTML pages in Navigation Bars, in Bridge Dialogs Boxes, or in the Content Pane.

  You cannot use ExtendScript features or Bridge DOM objects directly in an HTML page; for details, see Displaying HTML in Bridge.

Your script-defined windows or panes must use one or the other of these methods. You cannot mix ScriptUI controls with HTML controls.

Navigation Bars

Bridge provides two configurable navigation bars, one of which can be displayed at the top of the Bridge browser window (below the application navigation bar), and one at the bottom (above the status bar). There are two versions of each bar, for use with the two display modes of the Content pane. Access these existing NavBar Object objects through the Document Object' s properties.

- When the Content pane is displaying a web page (`Document.contentPaneMode="web"`), use these bars:

  ```
  topbar = app.document.navbars.web.top
  btmbar = app.document.navbars.web.bottom
  ```
- When the Content pane is displaying a folder's contents (`Document.contentPaneMode="filesystem"`), use these bars:

  ```
  topbar = app.document.navbars.filesystem.top
  btmbar = app.document.navbars.filesystem.bottom
  ```

The navigation bars are hidden by default. You can show and hide them by setting the `NavBar` object's visible property.

A navigation bar typically contains user-interface controls such as push buttons, radio buttons, scroll bars, list boxes, and so on. The `NavBar` objects are initially empty.

A navigation bar can display either ScriptUI user-interface controls that you add as children of the `NavBar` object, or an HTML page that you reference from the `NavBar` object. It cannot mix the two. In either case, you define the controls and program them to display information to or collect information from the user.

- Set the `NavBar.type` to `"scriptUI"` to display ScriptUI controls. See Displaying ScriptUI elements in a navigation bar.
- Set the `NavBar.type` to `"html"` to display HTML controls. See Displaying HTML in a Navigation Bar.

Dialogs Boxes

A dialog box, like a navigation bar, can display either ScriptUI controls or HTML controls, but not both. In the case of dialogs, there are two different types of objects.

- Create a ScriptUI Window Object to display ScriptUI controls. See Displaying ScriptUI Dialogs.
- Create a Bridge DOM Dialog Object to display HTML controls. See Displaying HTML in Bridge Dialogs.

Content Pane

The Content pane display is determined by the current thumbnail. When you select a thumbnail in the Favorites or Folders pane (or invoke a Thumbnail Object's open function), that Thumbnail Object's displayMode property determines what the Content pane displays.

You can define your own browse scheme and thumbnails that use that browse scheme, and add your thumbnails to the Favorites pane. (For details, see Script-Defined Browse Schemes.) When you do this, you can set the Thumbnail Object's displayMode property so that when the user selects it in the Favorites pane, it displays a user-interface panel in the Content pane.

- If you set `displayMode` to `web`, the Content pane shows the HTML page referenced by `displayPath`. You can define an HTML page containing user-interface controls. See Displaying HTML in the Content Pane

ScriptUI User Interfaces

ScriptUI is a module that defines windows and user-interface controls. There are three ways to display ScriptUI elements:

- You can create an independent ScriptUI window, populate it with ScriptUI controls, and invoke it from your script using the window's show function. See Displaying ScriptUI Dialogs
- You can add ScriptUI controls to the existing Navigation Bars, and display them by setting by setting the `NavBar` object's visible property to `true`.
- You can add ScriptUI controls to the UI panel associated with the Content pane, and display that panel by double-clicking a thumbnail whose displayMode property is set to `"script"`.

Displaying ScriptUI Dialogs

A script can define a window entirely in ScriptUI, by creating a Window Object and populating it with ScriptUI controls using its add method.

You can invoke a ScriptUI window from a script as a *modal* or *nonmodal* dialog.

- A modal dialog retains the input focus, and does not allow the user to interact with any other windows in the application (in this case, the Bridge browser window) until the dialog is dismissed. The function that invokes it does not return until the dialog is dismissed.
- A nonmodal dialog (known in ScriptUI as a *palette*), does not keep the input focus. The user can interact with the Bridge browser window while the dialog is up. The function that invokes it returns immediately, leaving the dialog on screen until the user or script closes it.

In ScriptUI, a modal dialog is a window of type `dialog`, and a modeless dialog is a window of type `palette`.

- Invoke a `dialog`-type window as a modal dialog using the `window` object's show function. In this case, the function does not return until the user dismisses the dialog, or you close it from a control's callback using the window's hide or close function. The `close` function allows you to return a value, which is passed to and returned from the call to `show`.
- Invoke a `palette`-type window as a modeless dialog using the `window` object's show function, which returns immediately, leaving the window on screen. The user can close the window using the OS-specific close icon on the frame, or you can close it from the script or a control's callback using the window's hide function.

The usage of the ScriptUI objects is discussed fully in Chapter 5, "Using ScriptUI," and complete syntax details are provided in Chapter 8, "ScriptUI Object Reference."

Displaying ScriptUI elements in a navigation bar

To display ScriptUI controls, set the `type` property to `"scriptui"`, then use the NavBar Object's add method to add controls. This is the same as the ScriptUI Window Object's add method.

➤ Example: Adding a path bar

This script defines a top navigation bar that uses ScriptUI objects to display a text field in which the user can type a path. When the user clicks **Go** or **New** in the navigation bar, the button's onClick handler navigates to that path.

```
// Change current document to browse the path in the path bar
//  (called when the Go button is clicked)
```

```
function browseToPath( message ) {
   try {
      var topbar = app.document.navbars.filesystem.top
      app.document.thumbnail =
         new Thumbnail( topbar.pathPanel.browseField.text );
   } catch ( error ) {
      Window.alert (error );
   }
}

// Open a new window to browse the path on the path bar.
//  (called when the New button is clicked)
function browseToPathNewWindow( message ) {
   try {
      var topbar = app.document.navbars.filesystem.top
      app.browseTo(topbar.pathPanel.browseField.text );
   } catch (error ) {
      Window.alert (error);
   }
}

// Create the PathBar panel on a nav bar
function addTopBarPanel(bar) {
   bar.type = "scriptui" // this is the default, so not really needed
   bar.pathPanel = bar.add( "panel", [5,5, 830, 35], undefined);
   bar.pathPanel.browseField = bar.pathPanel.add( "edittext",
      [3, 3, 700, 22],"");
   //theBrowseField = bar.pathPanel.browseField;
   bar.pathPanel.BrowseBtn = bar.pathPanel.add( "button",
      [710, 3, 760, 22],"Go");
   bar.pathPanel.BrowseBtnNewWin = bar.pathPanel.add( "button",
      [770, 3, 820, 22],"New");
   bar.pathPanel.BrowseBtn.onClick = browseToPath;
   bar.pathPanel.BrowseBtnNewWin.onClick = browseToPathNewWindow;
}

// Create the PathBar in the top navbar of the current document
var topbar = app.document.navbars.filesystem.top;
addTopBarPanel(topbar);
topbar.visible = true;
topbar.pathPanel.browseField.text = theDocument.thumbnail.path;
```

For detailed information on using the ScriptUI objects, see Chapter 5, "Using ScriptUI," and Chapter 8, "ScriptUI Object Reference."

Displaying HTML in Bridge

There are three mechanisms you can use to display an HTML UI within Bridge:

- A top or bottom `NavBar` displays HTML when `navBar.file` is set to the path of the HTML file, and `navBar.type="html"`.
- A `Dialog` object always displays HTML UI controls (as opposed to a ScriptUI dialog object, which displays ScriptUI controls). You specify the HTML file to display as the argument when creating the Dialog object. For example:

```
var myDialog = new Dialog("/C/BridgeScripts/HTML/dialogUI.html");
```

- When you set `thumbnail.displayPath` to the path of an HTML file, and `thumbnail.displayMode= "web"`, then selecting that thumbnail in the Folders or Favorites pane displays the HTML page in the Content pane.

In order to display the HTML, Bridge opens an embedded browser, which runs a standard JavaScript engine in a different process from the Bridge ExtendScript engine. The standard JavaScript engine can access only the standard HTML DOM. A script on the HTML page cannot directly access the Bridge DOM, or make use of ExtendScript features such as localization.

For a script in your UI page to communicate with the Bridge DOM, the HTML JavaScript engine and the Bridge ExtendScript engine must exchange values via *remote calls*.

- For the JavaScript code to make remote calls to ExtendScript, you define *callback* functions on the Bridge object, and invoke them from the HTML page with the JavaScript `call` function. The callback functions access Bridge objects on the Bridge side and pass values back to the HTML page. See Defining callbacks for HTML scripts.
- Your HTML page can define its own JavaScript functions in a script. For the Bridge side to use these functions, it must make a remote call using the Bridge object's `execJS` function. See Executing script functions defined on HTML UI pages.

The three mechanisms for displaying an HTML UI differ slightly in the details of how you define and pass callbacks and invoke script-defined functions. This section provides examples for a web page displayed in the Content pane, in response to selecting a web-type thumbnail. Examples for navigation bars and dialogs are given with the discussions of those objects above.

When you make remote calls, you can pass simple values such as strings and numbers directly. However, in order to pass complex values such as objects and arrays, you must deconstruct them on the passing side using the JavaScript function `toSource`, and reconstruct them on the receiving side using the JavaScript function `eval`. Examples are given for callbacks; see Passing Complex Values in Remote Calls. The embedded browser does not support toSource, so you cannot pass complex values from the HTML page back to the Bridge ExtendScript engine.

Defining callbacks for HTML scripts

If you want to make use of Bridge DOM values to dynamically alter the HTML controls as the user works with them, you must make calls back to the Bridge DOM through a set of callbacks that you define. The

exact way that you define and store the callbacks depends on which of the HTML mechanisms you are using:

- Defining callbacks for a dialog

  For a dialog, you define callback functions in a structure that you pass to the `Dialog` object's open or run function when you invoke the dialog. The syntax for the callbacks argument is:

  ```
  {
   fn_name1: function( args ) { fn1_definition },
   fn_name2: function( args ) { fn2_definition }
  }
  ```

  The dialog's HTML page can invoke these functions using the JavaScript `call` method. See the Using callbacks in an HTML dialog.

- Defining callbacks for the Content pane or navigation bar

  For HTML displayed in a navigation bar or in the Content pane, you define callback functions in the `jsFuncs` property of the appropriate object:

  - When a Thumbnail Object displays an HTML page in the Content pane, the Document Object's jsFuncs property stores callback functions for that page. See the examples given below.
  - For a page displayed in a navigation bar, the callbacks are stored in theNavBar Object's jsFuncs property. For examples, see Displaying HTML in a Navigation Bar.

From the HTML page, you can invoke your defined callback functions using the JavaScript `call` function. Typically, you will do this from a control's event handler, such as the `onClick` method for a button. For example, suppose one of your callbacks is defined as:

```
{ myCB: function(x) { return x > 0 } }
```

This defines a function named `myCB`. Within the HTML page's JavaScript, invoke the `myCB` Bridge DOM method as follows:

```
var positive = call("myCB", 29);
```

You must use the JavaScript `call` method to invoke callback functions. You cannot simply invoke them by name.

A callback function can access the Bridge DOM and pass back a response, as shown in the examples. The callback functions can receive and return simple types directly, but must use `eval` to reconstruct complex types passed as arguments from the HTML side, and use `toSource` to serialize complex types that you wish to return. See Passing Complex Values in Remote Calls.

Executing script functions defined on HTML UI pages

An HTML page that displays user-interface controls within Bridge can itself contain a script that defines functions. The `execJs` method (defined on the `Document`, `NavBar`, and `Dialog` objects) allows a Bridge script to invoke a JavaScript method defined in an HTML page.

- When a Thumbnail Object displays an HTML UI in the Content pane, use the Document Object's execJS method to execute functions defined in the script for that page. See the example below.
- For a page displayed in a navigation bar, use the NavBar Object's execJS method to execute functions defined in the script for that page.
- For a modeless HTML dialog, use the Dialog Object's execJS method to execute functions defined in the script for the dialog.

You should make sure that the HTML page which defines the remote function is actually loaded before you invoke the remote function with `execJS`.

Caution: You cannot call the `execJS` method from within a callback function. Doing so causes Bridge to hang. For an alternative, see Scheduling tasks from callbacks.

The `execJS` method takes as its argument a string that contains the entire function call to be executed. For example, this JavaScript code packages a call to the function `updatePath`, defined in the HTML displayed by `myDialog`:

```
myDialog.execJS("updatePath('" + escape(tn.path) + "')");
```

In this case, it is passing a pathname that contains the backslash (\), which is an escape character. It uses the `escape` function to create the argument string, and on the HTML side, the `updatePath` function uses `unescape` to retrieve the path from the argument string:

```
<script> //define fns to be called from Bridge
  function updatePath(path) { window.path.value = unescape(path) };
</script>
```

For a more complete version of this example, see Calling functions defined in an HTML navigation bar. The technique is exactly the same for a dialog as for a navigation bar, except for calling the function in the dialog object.

Displaying HTML in Bridge Dialogs

The Dialog Object represents a window that displays an HTML page. Use the open function to open the window as a modeless dialog. This function returns immediately, and the dialog remains on screen until the user or script dismisses it.

The `open` function takes as an argument a set of callback functions. These callbacks are used to respond to a dialog-closing event, and to provide the dialog's HTML JavaScript code with access to the Bridge DOM (see Communicating with Bridge from dialog JavaScript).

You can provide special callback functions named `willClose` and `onClose`. If they are provided, these are called automatically when the dialog closes in response to a user action (such as clicking the window's close icon, or clicking your button that sets `closing` to `true`).

- `willClose` takes no arguments and returns a Boolean value. If it returns `true`, Bridge calls the `onClose` callback (if provided). If it returns `false`, the closing operation is aborted.
- `onClose` takes no arguments and returns `undefined`. It should perform any cleanup operations for the dialog and its associated code.

Communicating with Bridge from dialog JavaScript

The HTML page displayed in a dialog runs its own JavaScript engine, which has access only to the HTML DOM. If the page needs to exchange data with your Bridge DOM objects, you must use the remote call mechanisms.

- You can define callbacks and pass them to the dialog as arguments to the run or open function that invokes it. You can call them remotely from the HTML page's JavaScript using the JavaScript `call` function. These callback provide access to the Bridge DOM objects. See Using callbacks in an HTML dialog.

- You can define JavaScript functions in the HTML page's script, and call them remotely from Bridge using the Dialog Object's execJS function—as long as you do not call them directly from a callback. See Calling functions defined in an HTML dialog and Scheduling tasks from callbacks.

Simple values such as strings and numbers can simply be passed back and forth, as shown in the examples below. Complex value such as arrays or objects must be deconstructed and reconstructed, using `toSource` and `eval`. For details of how to pass objects in remote function calls, see Passing Complex Values in Remote Calls.

Using callbacks in an HTML dialog

The callback functions that you define for a dialog are available to the code in the HTML page, which can invoke them using the `call` function. They run in Bridge's JavaScript engine, and can use Bridge DOM objects.

For example, suppose the `callback` argument that you pass to the open function has the value:

```
{ isGreater: function(x) { return x > myDialog.height } }
```

A method in the HTML page (an event handler, for instance) can invoke the function and receive the result as follows:

```
var newHeightOK =  call("isGreater", 29);
```

The following example creates a dialog that references an HTML page, which provides interface controls to select a file and a metadata key, and a button to request the metadata from the Bridge DOM.

Clicking the **Get Metadata** button invokes the Bridge DOM `getMetadata` callback, which is defined in the `callbacks` argument to the `open` or `run` method that invokes the dialog. The `getMetadata` function retrieves the metadata for the specified file by creating and accessing a Thumbnail object for that file. It sends the results back to the HTML JavaScript, which displays those results in the a text-box control on the HTML page.

The user can close the dialog box by clicking the **Done** button, which sets the dialog's `closing` property to `true`. This in turn invokes the `doClose` callback, which is also defined in the `callbacks` argument. The `doClose` callback allows you to take any cleanup actions necessary prior to the closing of the dialog box. In this case, it simply demonstrates that it has been called by putting up an alert message.

Code in the HTML file

This HTML page is displayed by the two dialog examples below. This code, in the file `C:/BridgeScripts/HTML/dialogUI.html`, defines the HTML controls, and includes event-handling code that invokes the dialog's callback functions.

```
<HTML>
<HEAD>
<META http-equiv="Content-Script-Type" content="text/javascript">
<TITLE>Bridge UI/HTML Integration</TITLE>
</HEAD>
<BODY>

For File: <INPUT type=file name="fname" size = "65">
<br><br>
Get Metadata key:
   <select name="mdkey">
      <option value="FileSize">FileSize
```

```
      <option value="FileName">FileName
      <option value="DocumentKind">DocumentKind
    </select>
  <INPUT type=button title="Get Metadata" name="go" value="Get Metadata"
    onclick="var t = call('getMetadata',window.fname.value,
                                  window.mdkey.value);
      window.mresult.value = t;" >
  <br><br>
  Result: <input type=text title="mresult" name="mresult" size = "50">
  <br><br><br><br><br><br>
  <Input type=button title="Done" name="done" value="Done"
    onclick="var d = call('closeDialog');" >

  </BODY>
  </HTML>
```

Code in the Bridge script

This script defines the dialog and its callback functions, and invokes it using the open function, which opens it as a modeless dialog.

The `open` function returns immediately, demonstrated by the fact that the alert message, "Return from dialog.open", appears immediately after the dialog is opened. Although it is not demonstrated by this example, your Bridge script could close the dialog before the user dismisses it, using the `dialog.close` method. For example, you might want to close the dialog if the user opens a new Bridge Browser window, or as part of your cleanup in response to application shutdown. If you do this, the `doClose` callback is automatically executed.

```
myFile = File("/C/BridgeScripts/HTML/dialogUI.htm");
var mdDialog = new Dialog(myFile);
var jsCallbacks = {
  getMetadata: function(x,y) {
     var tn = new Thumbnail( File(x) );
     return(tn.metadata[y]);
     },
  closeDialog: function() {
     Window.alert("Closing Dialog" + mdDialog);
     mdDialog.closing = true;
     },
  doClose: function() {
     /* do any cleanup before closing window */
     /* called automatically by the dialog's close function */
     Window.alert("In doClose");
     }
  };
mdDialog.title = "Get Metadata";
mdDialog.width = 512;
mdDialog.height = 512;
mdDialog.modal = false;

mdDialog.open(jsCallbacks); // display as modeless dialog
Window.alert("Returned from dialog.open ");
```

Calling functions defined in an HTML dialog

If the HTML page displayed in the dialog defines any JavaScript functions of its own, Bridge can make remote calls to those functions using the Dialog Object's execJS function. For a modeless dialog, this works exactly the same way as for a navigation bar or a page displayed in the Content pane. For an example, see Calling functions defined in an HTML navigation bar.

Caution: You cannot call the `execJS` method from within a callback function. Doing so causes Bridge to hang. For an alternative, see Scheduling tasks from callbacks.

Displaying HTML in a Navigation Bar

To display HTML controls, set, set the `type` property to `"html"`, and the `file` property to the HTML file that defines the page you want to display.

When a navigation bar displays HTML, it runs its own JavaScript engine, which has access only to the HTML DOM. If the page needs to exchange data with your Bridge DOM objects, you must use the remote call mechanisms.

- You can define and store callbacks in the NavBar Object's jsFuncs property and call them remotely from the HTML page's JavaScript using the JavaScript `call` function. These callback provide access to the Bridge DOM objects. See Using callbacks from an HTML navigation bar.
- You can define JavaScript functions in the HTML page's script, and call them remotely from Bridge using the NavBar Object's execJS function—as long as you do not call them directly from a callback. See Calling functions defined in an HTML navigation bar and Scheduling tasks from callbacks.

These examples demonstrate HTML navigation bars that use remote calls in both directions.

Using callbacks from an HTML navigation bar

This example displays HTML controls in the top navigation bar, allowing the user to select a file and a metadata key, and to request the metadata value from the Bridge DOM by clicking a button. Clicking the button invokes the `getMetadata` callback function, which is defined and stored in the NavBar Object's jsFuncs property.

When the callback is executed, it retrieves the metadata for the specified file, and sends the results back to HTML JavaScript. The HTML code displays the result in another control.

Code in the Bridge script

This code defines the navigation bar, and a callback to be invoked remotely from the HTML page.

```
var jsCallbacks = {
   getMetadata: function(x,y) {
      var tn = new Thumbnail( File(x) );
      return(tn.metadata[y]);
   },
};

function addNavBar() {
   var topbar = app.document.navbars.filesystem.top;
   topbar.height = 65;
   topbar.type = "html";
   topbar.file = File("/C/BridgeScripts/HTML/navbarUI.html");
```

```
    topbar.jsFuncs = jsCallbacks;
    topbar.visible = true;
  }

  addNavBar();
```

Code in the HTML file

This code defines the controls to be displayed in the bar and invokes the `getMetadata` callback function in response to a button click.

```
<HTML>
<HEAD>
<META http-equiv="Content-Script-Type" content="text/javascript">
<TITLE>Bridge UI/HTML Integration</TITLE>
</HEAD>
<BODY>

For File: <INPUT type=file name="fname" size = "100">    <br>
Get Metadata key:
   <select name="mdkey">
      <option value="FileSize">FileSize
      <option value="FileName">FileName
      <option value="DocumentKind">DocumentKind
      <option value="Creator">Creator
   </select>
<INPUT type=button title="Get Metadata" name="go" value="Get Metadata"
   onclick="var t = call('getMetadata',window.fname.value,
                        window.mdkey.value);
   window.mresult.value = t;" >
Result: <input type=text title="mresult" name="mresult" size = "50">

</BODY>
</HTML>
```

Calling functions defined in an HTML navigation bar

If the HTML page displayed in a navigation bar defines any JavaScript functions of its own, Bridge can make remote calls to those functions using the NavBar Object's execJS function, as shown in the following example. If a Bridge HTML dialog displays the HTML page, use the same technique, but call the Dialog Object's execJS function.

In this example, the HTML page displayed in the top navigation bar has two text boxes, a path and a quantity. Each time the user selects a `Thumbnail` for a file or folder, the corresponding path name is displayed in the **Path** box. If the user selects a folder, the number of items in the folder is displayed in the **Number of Items** box.

The HTML script defines two JavaScript functions, `updatePath` and `updateItemQty`, to perform these updates. However, these functions operate on values retrieved from Bridge objects, which must be passed in from the Bridge ExtendScript engine.

The update functions need to be invoked in response the user selecting a thumbnail. This action generates a `Thumbnail select` event on the Bridge side, so the Bridge script defines a handler for that event, which invokes the remote functions by calling `execJS`. It gets the required values from Bridge DOM

objects, packages the function call (the name and its argument list) as a string, and passes that string as the argument to `execJS`.

Before you use `execJS` to call functions defined in the HTML JavaScript code, you need to make sure that the page is loaded, so that the functions are defined when you call them. In this example, the page uses the HTML/JavaScript `onloaded` event defined on the `BODY` tag to invoke a callback (defined in the `NavBar`'s `jsFuncs` property), which sets a Bridge-script global variable, `htmlLoaded`. The event handler checks that variable before calling the remote functions.

If the page is loaded, the event handler invokes the HTML JavaScript function `updatePath`, which writes the path argument into the **Path** text box. If the event happened to a folder object, it invokes the HTML JavaScript function `updateItemQty`, which writes the `itemQty` argument to the **Number of Items** text box. If the event was triggered by a noncontainer object, the script writes the string `"n/a"` to the text box.

Note: JavaScript uses the backslash (\) as the escape character, but the backslash is part of Windows platform path names. Therefore, in order to pass the path name value, the script uses the JavaScript `escape` function to encode the name it sends to HTML. On the HTML JavaScript side, `unescape` decodes the string so it is properly displayed in the **Path** box with the backslash character.

Code in the Bridge script

The Bridge script points the navigation bar to the HTML page, and invokes functions defined on that page.

```
var htmlLoaded = false;
var jsCallbacks = {
   loaded: function() {
      htmlLoaded = true; //set a global to make sure page is loaded
   }
};
function addNavBar() {
   var topbar = app.document.navbars.filesystem.top;
   topbar.height = 65;
   topbar.type = "html";
   topbar.file = File("/C/BridgeScripts/HTML/navbar2UI.html");
   topbar.jsFuncs = jsCallbacks;
   topbar.visible = true;
}

onEvent = function( event ) {
   try {
      if( event.object.constructor.name == "Thumbnail") {
         if( event.type == "select" && htmlLoaded) {
         // for thumbnail selection
            var tn = event.object;
            var topNb = app.document.navbars.filesystem.top;
            //call fns defined in HTML page's script
            topNb.execJS("updatePath('" + escape(tn.path) + "')");
            if(tn.container) {
               topNb.execJS("updateItemQty(" + tn.children.length + ")");
            }
            else{
               topNb.execJS("updateItemQty('n/a')");
            }
         }
      }
   } catch (error) {
```

```
            alert (error);
        }
    }
    // Add the event handler to the application
    app.eventHandlers.push( { handler: onEvent} );
    addNavBar(); //show the navigation bar
```

Code in the HTML file

The displayed HTML page defines two functions, which are invoked from the Bridge script.

```
<HTML>
<HEAD>
<META http-equiv="Content-Script-Type" content="text/javascript">
<TITLE>Bridge UI/HTML Integration</TITLE>
<script>
  //define fns to be called from Bridge
  function updatePath(path) { window.path.value = unescape(path) };
  function updateItemQty(qty) { window.itemQty.value = qty };
</script>
</HEAD>

<BODY onload = "call('loaded')">
  <!--use callback to set Bridge global-->
Path: <INPUT type=text name="path" size = "100">    <br>
Number of Items: <input type=text title="itemQty" name="itemQty" size =
"25">

</BODY>
</HTML>
```

Displaying HTML in the Content Pane

The following examples use the `Thumbnail` and Content pane mechanism, and demonstrate how to pass simple and complex values between Bridge and an HTML UI page via callback functions. For similar examples that use the `NavBar` and `Dialog` mechanisms, see Displaying HTML in a Navigation Bar and Using callbacks in an HTML dialog.

Callback example: Requesting specific metadata value for a file

This example shows how to use an HTML file as a user-interface to the Bridge DOM.

- The Bridge DOM script creates a Thumbnail Object that uses a script-defined browse scheme. Its `displayPath` is set to the path of the HTML file shown below, and its `displayMode` set to `web`. This thumbnail is added to the Favorites pane.
- When the user selects the "HTML Sample" icon in the Folders pane, the `testUI.html` page appears in the Content pane. This HTML file provides interface controls to select a file and a metadata key, and a button to request the metadata from the Bridge DOM.
- Clicking the button invokes the Bridge DOM `getMetadata` function, which is defined in the `Document.jsFuncs` property. The `getMetadata` function retrieves the metadata for the specified file, and sends the results back to HTML JavaScript, which displays the results on the HTML page.

Code in the Bridge DOM script

```
var jsCallbacks = {
   getMetadata: function(x,y) {
      var tn = new Thumbnail( File(x) );
      return(tn.metadata[y]);
   },
};
app.document.jsFuncs = jsCallbacks;

app.registerBrowseScheme( "myScheme" );
var t = new Thumbnail("myScheme://root");
t.name = "HTML Sample";
t.displayMode = "web";
t.displayPath = "/C/myScripts/html/testUI.html";
app.favorites.insert(t); //add the thumbnail to Favorites
```

Code in the HTML file testUI.html

```
<HTML>
<HEAD>
<META http-equiv="Content-Script-Type" content="text/javascript">
<TITLE>Bridge UI/HTML Integration</TITLE>
</HEAD>
<BODY>

For File: <INPUT type=file name="fname" size = "65">
<br><br>
Get Metadata key:
   <select name="mdkey">
      <option value="FileSize">FileSize
      <option value="FileName">FileName
      <option value="DocumentKind">DocumentKind
   </select>
<INPUT type=button title="Get Metadata" name="go" value="Get Metadata"
   onclick="var t = call('getMetadata',window.fname.value,
                                 window.mdkey.value);
               window.mresult.value = t;" >
<br><br>
Result: <input type=text title="mresult" name="mresult" size = "50">
<br><br>

</BODY>
</HTML>
```

Passing Complex Values in Remote Calls

To exchange simple values such as strings and numbers between Bridge and an HTML UI page, you can simply pass arguments and return values of those types in your callback and `execJS` functions. However, complex values such as objects and arrays must be broken down and reconstructed on the other side. This is true for communication in both directions—callbacks from HTML to Bridge, and execution of HTML script functions by Bridge using `execJs`.

For a callback to receive an object as an argument, the calling function on the HTML side must serialize the object into a string, using `toSource`, and pass the serialized string. On the Bridge side, the callback

function uses `eval` to reconstruct the object from the serialized string. Similarly, to pass an object back, the callback function must use `toSource` to serialize the object and return the serialized string. The receiving code on the HTML side must in turn reconstruct the object using `eval`. See the example below.

Note: The embedded browser does not support the `toSource` method, so you cannot use this mechanism to pass complex values to and from HTML-page functions that you invoke using the `execJS` method. Pass only simple values from the HTML JavaScript engine to the Bridge ExtendScript engine.

Passing an object from Bridge to HTML/JavaScript

This example shows a Bridge Document Object callback that sends an object to a script on an HTML page being displayed in the Content pane.

Code in the Bridge DOM script

The callback function creates a JavaScript object and copies properties and values into it from file metadata, which it accesses through a `Thumbnail` object. It then uses `toSource` to create a serialized string from the object, and returns that string.

```
var jsCallbacks = {
  getFileInfo: function(x) {
     var tn = new Thumbnail( File(x) );
     var info = {};
     info.fileName = tn.metadata.FileName;
     info.fileSize = tn.metadata.FileSize;
     info.fileKind = tn.metadata.DocumentKind;
     return(info.toSource());
     }
};
app.document.jsFuncs = jsCallbacks;

app.registerBrowseScheme( "myScheme" );
var t = new Thumbnail("myScheme://root");
t.name = "HTML Sample";
t.displayMode = "web";
t.displayPath = "/C/MyScripts/HTML/contentPaneUI.html";
app.favorites.insert(t); // add the thumbnail to Favorites
```

Code in the HTML page

The HTML code makes a call to the callback function. When it receives the string value, it reconstitutes the object using `eval`.

```
<HTML>
<HEAD>
<META http-equiv="Content-Script-Type" content="text/javascript">
<TITLE>Bridge UI/HTML Integration</TITLE>
</HEAD>
<BODY>

For File: <INPUT type=file name="fname" size = "65">
<br><br>
<INPUT type=button title="Get File Info" name="go" value="Get File Info"
  onclick="var ret = call('getFileInfo',window.fname.value);
        var info = eval(ret);
```

```
          window.filename.value = info.fileName;
          window.filesize.value = info.fileSize;
          window.filekind.value = info.fileKind;" >
<br><br>
File Name: <input type=text title="filename" name="filename" size = "50">
<br><br>
File Size: <input type=text title="filesize" name="filesize" size = "50">
<br><br>
Doc Kind:  <input type=text title="filekind" name="filekind" size = "50">
<br><br>

</BODY>
</HTML>
```

Scheduling tasks from callbacks

You cannot call the `execJS` method from within a callback function (either stored in a `jsCallbacks` property or passed as an argument to the Dialog `open` or `run` method). This attempts to re-enter the JavaScript engine, which is already running and is not re-entrant. If you try to do this, Bridge will hang.

The alternative is to schedule a task, using the App Object's scheduleTask function, from within the callback function. From the function associated with the task, you can call `execJS`. Because it is not executed until the callback returns, the task is free to make another remote call.

The first argument to `scheduleTask` is a string containing a script—in this case, a call to the `execJS` function. For example:

```
var result = app.scheduleTask("myFn(3);", 10);
```

If the script itself contains any strings, those must be indicated by enclosed quotes. For example:

```
var result = app.scheduleTask("myFn('string argument');", 10);
```

If the enclosed string contains values derived from expressions, the script string must be concatenated, and can become quite complex:

```
var result = app.scheduleTask("myFn('" + escape(tn.path) + "')", 10);
```

The argument to `execJS` is also a string containing a script—in this case, a call to a function defined on an HTML page. When the arguments to that function are also strings, and those contain values derived from expressions, the resulting string is very complex.

The example below makes this string a little more manageable by breaking it down into modular pieces. First, it builds the argument string for the remote function:

```
var toRecordArg = "'For File: " + tn.metadata.FileName + "'";
```

It uses that to build the entire string for the remote function call, which is the argument to `execJS`:

```
var execFn = "recordData(" + toRecordArg + ")";
```

Building the string for the call to `execJS` requires an additional layer of embedded quotes, which is very difficult to achieve with only two types of quote character. To get around this, the example creates a variable for a string containing the double-quote character, and uses it to build the entire function call string, which is passed to `scheduleTask`:

```
var quote = '"';
app.scheduleTask("app.document.execJS(" + quote + execFn + quote + ")")
```

Scheduling a remote function execution

This example defines a thumbnail that uses a script-defined browse scheme and displays a web page. That web page (shown below) defines a function `recordData`. The page makes a call to a callback function defined in the script, and the callback function constructs and schedules calls to execJS, to be executed after the callback returns.

Bridge Script

The callbacks for the page (defined in the Document Object's jsFuncs property) include a function `getMetadata`, which schedules two calls to the `recordData` function. For each task, it builds a string containing a call to the Document Object's execJS function, and passes it to the `app.scheduleTask` function. (This is a very complex string, since the argument to `execJS` is also a string containing a complete function call to `recordData`, which itself takes a string argument.)

After the callback has returned, the scheduled tasks execute the remote function, which writes out data to one of the HTML page's controls.

```
htmlLoaded = false; // make sure page is loaded
// Define callback functions
var jsCallbacks = {
   getMetadata: function(x,y) {
      var quote = '"';
      var tn = new Thumbnail( File(x) );
      var toRecordArg = "'For File: " + tn.metadata.FileName + "'";
      var execFn = "recordData(" + toRecordArg + ")";
      app.scheduleTask("app.document.execJS("
            + quote + execFn + quote + ")",0, false);
      toRecordArg = "'    Metadata key " + y + " has value "
            + tn.metadata[y] + "'";
      execFn = "recordData(" + toRecordArg + ")";
      app.scheduleTask("app.document.execJS(" + quote + execFn
            + quote + ")",10, false);
      return(tn.metadata[y]);
   },
   loaded: function() {
      htmlLoaded = true; //page is loaded
      }
   };

app.document.jsFuncs = jsCallbacks;

// define thumbnail to show the HTML page
app.registerBrowseScheme( "myScheme" );
var t = new Thumbnail("myScheme://root");
t.name = "HTML Sample";
t.displayMode = "web";
t.displayPath = "/C/myScripts/HTM/testUISchedTask.html";

app.favorites.add(t); // add the thumbnail to the Favorites pane
app.document.thumbnail= t; // browse to the thumbnail
```

HTML code

This page, which is displayed by the thumbnail, defines a script function, `recordData`. The button's `onClick` method calls the `getMetadata` callback, which schedules the `recordData` function to be executed remotely after the callback has returned.

```
<HTML>
<HEAD>
<META http-equiv="Content-Script-Type" content="text/javascript">
<TITLE>Bridge UI/HTML Integration</TITLE>

<script>
function recordData(logString) {
  window.dataLog.value = window.dataLog.value + '\n' + unescape(logString);
  }
</script>

</HEAD>
<BODY onload="call('loaded')">

For File: <INPUT type=file name="fname" size = "65">
<br><br>
Get Metadata key:
  <select name="mdkey">
     <option value="FileSize">FileSize
     <option value="FileName">FileName
     <option value="DocumentKind">DocumentKind
  </select>
<INPUT type=button title="Get Metadata" name="go" value="Get Metadata"
  onclick="var t = call('getMetadata',window.fname.value,
                              window.mdkey.value);
             window.mresult.value = t;" >
  <br><br>
Result: <input type=text title="mresult" name="mresult" size = "50">
  <br><br>
DataLog: <br>
  <textarea type="dataLog" name = "dataLog" rows = "15"
     cols = "75"></textarea>
  <br><br>

</BODY>
</HTML>
```

4 Using File and Folder Objects

Overview

Because path name syntax is very different in Windows®, Mac OS®, and UNIX®, Adobe ExendScript defines the `File` and `Folder` objects to provide platform-independent access to the underlying file system. A File Object represents a disk file; a Folder Object represents a directory or folder.

- The `Folder` object supports file system functionality such as traversing the hierarchy; creating, renaming, or removing files; or resolving file aliases.
- The `File` object supports input/output functions to read or write files.

There are several ways to distinguish between a `File` and a `Folder` object. For example:

```
if (f instanceof File) ...
if (typeof f.open == "undefined") ...// Folders do not open
```

`File` and `Folder` objects can be used anywhere that a path name is required, such as in properties and arguments for files and folders. For details about the objects and their properties and methods, see Chapter 7, "File and Folder Object Reference."

Note: When you create two `File` objects that refer to the same disk file, they are treated as distinct objects. If you open one of them for I/O, the operating system may inhibit access from the other object, because the disk file already is open.

Specifying Paths

When creating a `File` or `Folder` object, you can specify a platform-specific path name, or an absolute or relative path in a platform-independent format known as *universal resource identifier* (*URI*) notation. The path stored in the object is always an absolute, full path name that points to a fixed location on the disk.

- Use the `toString` method to obtain the name of the file or folder as string containing an absolute path name in URI notation.
- Use the fsName property to obtain the platform-specific file name.

Absolute and relative path names

An absolute path name in URI notation describes the full path from a root directory down to a specific file or folder. It starts with one or two slashes (`/`), and a slash separates path elements. For example, the following describes an absolute location for the file `myFile.jsx`:

```
/dir1/dir2/mydir/myFile.jsx
```

A relative path name in URI notation is appended to the path of the current directory, as stored in the globally available current property of the `Folder` class. It starts with a folder or file name, or with one of the special names dot (`.`) for the current directory, or dot dot (`..`) for the parent of the current directory. A

slash (/) separates path elements. For example, the following paths describe various relative locations for the file `myFile.jsx`:

Path	Location
`myFile.jsx` `./myFile.jsx`	In the current directory.
`../myFile.jsx`	In the parent of the current directory.
`../../myFile.jsx`	In the grandparent of the current directory.
`../dir1/myFile.jsx`	In `dir1`, which is parallel to the current directory.

Relative path names are independent of different volume names on different machines and operating systems, and therefore make your code considerably more portable. You can, for example, use an absolute path for a single operation, to set the current directory in the `Folder.current` property, and use relative paths for all other operations. You would then need only a single code change to update to a new platform or file location.

Character interpretation in paths

There are some platform differences in how pathnames are interpreted:

- In Windows and Mac OS, path names are not case sensitive. In UNIX, paths are case sensitive.
- In Windows, both the slash (/) and the backslash (\) are valid path element separators.
- In Mac OS, both the slash (/) and the colon (:) are valid path element separators.

If a path name starts with two slashes (or backslashes in Windows), the first element refers to a remote server. For example, `//myhost/mydir/myfile` refers to the path `/mydir/myfile` on the server `myhost`.

URI notation allows special characters in pathnames, but they must specified with an escape character (%) followed by a hexadecimal character code. Special characters are those that are not alphanumeric and not one of the characters:

```
/ - — . ! ~ * ' ( )
```

A space, for example, is encoded as `%20`, so the file name `"my file"` is specified as `"my%20file"`. Similarly, the character ä is encoded as `%E4`, so the file name `"Bräun"` is specified as `"Br%E4un"`.

This encoding scheme is compatible with the global JavaScript functions `encodeURI` and `decodeURI`.

The home directory

A path name can start with a tilde (~) to indicate the user's home directory. It corresponds to the platform's `HOME` environment variable.

UNIX and Mac OS assign the `HOME` environment variable according to the user login. In Mac OS, the default home directory is `/Users/`*`username`*. In UNIX, it is typically `/home/`*`username`* or `/users/`*`username`*. Extend Script assigns the home directory value directly from the platform value.

In Windows, the `HOME` environment variable is optional. If it is assigned, its value must be a Windows path name or a path name referring to a remote server (such as `\\myhost\mydir`). If the `HOME` environment variable is undefined, the Extend Script default is the user's home directory, usually the `C:\Documents and Settings\`*`username`* folder.

Note: A script can access many of the folders that are specified with platform-specific variables through static, globally available Folder class properties; for instance, appData contains the folder that stores application data for all users.

Volume and drive names

A volume or drive name can be the first part of an absolute path in URI notation. The values are interpreted according to the platform.

Mac OS volumes

When Mac OS X starts, the startup volume is the root directory of the file system. All other volumes, including remote volumes, are part of the `/Volumes` directory. The `File` and `Folder` objects use these rules to interpret the first element of a path name:

- If the name is the name of the startup volume, discard it.
- If the name is a volume name, prepend `/Volumes`.
- Otherwise, leave the path as is.

Mac OS 9 is not supported as an operating system, but the use of the colon as a path separator is still supported and corresponds to URI and to Mac OS X paths as shown in the following table. These examples assume that the startup volume is `MacOSX`, and that there is a mounted volume `Remote`.

URI path name	Mac OS 9 path name	Mac OS X path name
`/MacOSX/dir/file`	`MacOSX:dir:file`	`/dir/file`
`/Remote/dir/file`	`Remote:dir:file`	`/Volumes/Remote/dir/file`
`/root/dir/file`	`Root:dir:file`	`/root/dir/file`
`~/dir/file`		`/Users/jdoe/dir/file`

Windows drives

In Windows, volume names correspond to drive letters. The URI path `/c/temp/file` normally translates to the Windows path `C:\temp\file`.

If a drive exists with a name matching the first part of the path, that part is always interpreted as that drive. It is possible for there to be a folder in the root that has the same name as the drive; imagine, for example, a folder `C:\C` in Windows. A path starting with `/c` always addresses the drive `C:`, so in this case, to access the folder by name, you must use both the drive name and the folder name, for example `/c/c` for `C:\C`.

If the current drive contains a root folder with the same name as another drive letter, that name is considered to be a folder. That is, if there is a folder `D:\C`, and if the current drive is `D:`, the URI path `/c/temp/file` translates to the Windows path `D:\c\temp\file`. In this case, to access drive C, you would have to use the Windows path name conventions.

To access a remote volume, use a uniform naming convention (UNC) path name of the form `//`*`servername`*`/`*`sharename`*. These path names are portable, because both Max OS X and UNIX ignore multiple slash characters. Note that in Windows, UNC names do *not* work for local volumes.

These examples assume that the current drive is `D:`

URI path name	Windows path name
/c/dir/file	c:\dir\file
/remote/dir/file	D:\remote\dir\file
/root/dir/file	D:\root\dir\file
~/dir/file	C:\Documents and Settings\jdoe\dir\file

Aliases

When you access an alias, the operation is transparently forwarded to the real file. The only operations that affect the alias are calls to `rename` and `remove`, and setting properties `readonly` and `hidden`. When a `File` object represents an alias, the `alias` property of the object returns `true`, and the `resolve` method returns the `File` or `Folder` object for the target of the alias.

In Windows, all file system aliases (called *shortcuts*) are actual files whose names end with the extension `.lnk`. Never use this extension directly; the `File` and `Folder` objects work without it.

For example, suppose there is a shortcut to the file `/folder1/some.txt` in the folder `/folder2`. The full Windows file name of the shortcut file is `\folder2\some.txt.lnk`.

To access the shortcut from a `File` object, specify the path `/folder2/some.txt`. Calling that `File` object's `open` method opens the linked file (in `/folder1`). Calling the `File` object's `rename` method renames the shortcut file itself (leaving the `.lnk` extension intact).

However, Windows permits a file and its shortcut to reside in the same folder. In this case, the `File` object always accesses the original file. You cannot create a `File` object to access the shortcut when it is in the same folder as its linked file.

A script can create a file alias by creating a `File` object for a file that does not yet exist on disk, and using its createAlias method to specify the target of the alias.

Portability issues

If your application will run on multiple platforms, use relative path names, or try to originate path names from the home directory. If that is not possible, work with Mac OS X and UNIX aliases, and store your files on a machine that is remote to your Windows machine so that you can use UNC names.

As an example, suppose you use the UNIX machine `myServer` for data storage. If you set up an alias share in the root directory of `myServer`, and if you set up a Windows-accessible share at `share` pointing to the same data location, the path name `//myServer/share/file` would work for all three platforms.

Unicode I/O

When doing file I/O, Adobe applications convert 8-bit character encoding to Unicode. By default, this conversion process assumes that the system encoding is used (code page 1252 in Windows or Mac Roman in Mac OS). The `encoding` property of a `File` object returns the current encoding. You can set the `encoding` property to the name of the desired encoding. The `File` object looks for the corresponding encoder in the operating system to use for subsequent I/O. The name is one of the standard Internet names that are used to describe the encoding of HTML files, such as `ASCII`, `X-SJIS`, or `ISO-8859-1`. For a complete list, see File and Folder Supported Encoding Names.

A special encoder, `BINARY`, is provided for binary I/O. This encoder simply extends every 8-bit character it finds to a Unicode character between 0 and 255. When using this encoder to write binary files, the encoder writes the lower 8 bits of the Unicode character. For example, to write the Unicode character `1000`, which is 0x3E8, the encoder actually writes the character 232 (0xE8).

The data of some of the common file formats (UCS-2, UCS-4, UTF-8, UTF-16) starts with a special byte order mark (BOM) character (`\uFEFF`). The `File.open` method reads a few bytes of a file looking for this character. If it is found, the corresponding encoding is set automatically and the character is skipped. If there is no BOM character at the beginning of the file, `open()` reads the first 2 KB of the file and checks whether the data might be valid UTF-8 encoded data, and if so, sets the encoding to UTF-8.

To write 16-bit Unicode files in UTF-16 format, use the encoding UCS-2. This encoding uses whatever byte-order format the host platform supports.

When using UTF-8 encoding or 16-bit Unicode, always write the BOM character "`\uFEFF`" as the first character of the file.

File Error Handling

Each object has an `error` property. If accessing a property or calling a method causes an error, this property contains a message describing the type of the error. On success, the property contains the empty string. You can set the property, but setting it only causes the error message to be cleared. If a file is open, assigning an arbitrary value to the property also resets its error flag.

For a complete list of supported error messages, see File and Folder Error Messages.

5 Using ScriptUI

Overview

ScriptUI is a component that works with the ExtendScript JavaScript interpreter to provide JavaScript programs with the ability to create and interact with user interface elements. It provides an object model for windows and UI control elements within an Adobe Creative Suite 2 application. ScriptUI objects are available to JavaScript scripts for the following applications:

- Adobe Photoshop CS2
- Adobe Bridge CS2

Note: Adobe GoLive® CS2 SDK includes another version of these objects, which have diverged somewhat in usage and functionality. See the *Adobe GoLive CS SDK Programmer's Guide* and *Adobe GoLive CS SDK Programmer's Reference* for details.

This chapter describe how to work with these objects, and Chapter 8, "ScriptUI Object Reference," provides the details of the objects with their properties, methods, and creation parameters.

ScriptUI Programming Model

ScriptUI defines `Window` objects that represent platform-specific windows, and various control elements such as `Button` and `StaticText`, that represent user-interface controls. These objects share a common set of properties and methods that allow you to query the type, move the element around, set the title, caption or content, and so on. Many element types also have properties unique to that class of elements.

Creating a window

ScriptUI defines the following types of windows:

- Modal dialog boxes: not resizeable, holds focus when shown.
- Floating palettes: also called modeless dialogs, not resizeable. (Photoshop CS2 does not support script creation of palette windows.)
- Main windows: resizeable, suitable for use as an application's main window. (Main windows are not normally created by script developers for Adobe Creative Suite 2 applications. Photoshop CS2 does not support script creation of main windows.)

To create a new window, use the `Window` constructor function. The constructor takes the desired type of the window. The type is `"dialog"` for a modal dialog, or `"palette"` for a modeless dialog or floating palette. You can supply optional arguments to specify an initial window title and bounds.

The following example creates an empty dialog with the variable name `dlg`, which is used in subsequent examples:

```
// Create an empty dialog window near the upper left of the screen
var dlg = new Window('dialog', 'Alert Box Builder', [100,100,480,490]);
```

Newly created windows are initially hidden; the `show` method makes them visible and responsive to user interaction. For example:

```
dlg.show();
```

Container elements

All `Windows` are containers—that is, they contain other elements within their bounds. Within a `Window`, you can create other types of container elements: `Panels` and `Groups`. These can contain control elements, and can also contain other `Panel` and `Group` containers. However, a `Window` cannot be added to any container.

- A `Group` is the simplest container used to visually organize related controls. You would typically define a group and populate it with related elements, for instance an `edittext` box and its descriptive `statictext` label.
- A `Panel` is a frame object, also typically used to visually organize related controls. It has a text property to specify a title, and can have a border to visually separate the collection of elements from other elements of a dialog.

You might create a `Panel` and populate it with several `Groups`, each with their own elements. You can create nested containers, with different layout properties for different containers, in order to define a relatively complex layout without any explicit placement.

You can add elements to any container using the `add` method (see Adding elements to containers). An element added to a container is considered a child of that container. Certain operations on a container apply to its children; for example, when you hide a container, its children are also hidden.

Window layout

When a script creates a `Window` and adds various UI elements to it, the locations and sizes of elements and spacing between elements is known as the *layout* of the window. Each UI element has properties which define its location and dimensions: `location`, `size`, and `bounds`. These properties are initially undefined, and a script that employs Automatic Layout should leave them undefined for the main window as well as its contained elements, allowing the automatic layout mechanism to set their values.

Your script can access these values, and (if not using auto-layout) set them as follows:

- The `location` of a window is defined by a Point object containing a pair of coordinates (`x` and `y`) for the top left corner (the *origin*), specified in the screen coordinate system. The `location` of an element within a window or other container is defined as the origin point specified in the container's coordinate system. That is, the `x` and `y` values are relative to the origin of the container.

  The following examples show equivalent ways of placing the content region of an existing window at screen coordinates [10, 50]:

  ```
  win.location = [10, 50];
  win.location = {x:10, y:50};
  win.location = "x:10, y:50";
  ```

- The `size` of an element's region is defined by a Dimension object containing a `width` and `height` in pixels.

  The following examples show equivalent ways of changing an existing window's width and height to 200 and 100:

  ```
  win.size = [200, 100];
  win.size = {width:200, height:100};
  win.size = "width:200, height:100";
  ```

  This example shows how to change a window's height to 100, leaving its location and width unchanged:

  ```
  win.size.height = 100;
  ```

- The `bounds` of an element are defined by a Bounds object containing both the origin point (`x`, `y`) and size (`width`, `height`) To define the size and location of windows and controls in one step, use the `bounds` property.

  The value of the `bounds` property can be a string with appropriate contents, an inline JavaScript `Bounds` object, or a four-element array. The following examples show equivalent ways of placing a 380 by 390 pixel window near the upper left corner of the screen:

  ```
  var dlg = new Window('dialog', 'Alert Box Builder', [100,100,480,490]);
  dlg.bounds = [100,100,480,490];
  dlg.bounds = {x:100, y:100, width:380, height:390};
  dlg.bounds = {left:100, top:100, right:480, bottom:490};
  dlg.bounds = "left:100, top:100, right:480, bottom:490";
  ```

The `window` dimensions define the size of the *content region* of the window, or that portion of the window that a script can directly control. The actual window size is typically larger, because the host platform's window system typically adds title bars and borders. The `bounds` property for a `Window` refers only to its content region. To determine the bounds of the frame surrounding the content region of a window, use the Window Object's frameBounds property.

Adding elements to containers

To add elements to a `window`, `panel`, or `group`, use the container's add method. This method accepts the `type` of the element to be created and some optional parameters, depending on the element type. It creates and returns an object of the specified type.

In additions to windows, ScriptUI defines the following user-interface elements and controls:

- Panels (frames) and groups, to collect and organize other control types
- Push buttons with text or icons, radio buttons, checkbox buttons
- Static text or images, edit text
- Progress bars, scrollbars, sliders
- Lists, which include list boxes and drop-down (also called popup) lists. Each item in a list is a control of type `item`, and the parent list's `items` property contains an array of child items. You can add list items with the parent list's add method.

You can specify the initial size and position of any new element relative to the working area of the parent container, in an optional `bounds` parameter. Different types of elements have different additional parameters. For elements which display text, for example, you can specify the initial text. See the ScriptUI Object Reference for details.

The order of optional parameters must be maintained. Use the value `undefined` for a parameter you do not wish to set. For example, if you want to use automatic layout to determine the bounds, but still set the title and text in a panel and button, the following creates `Panel` and `Button` elements with an initial `text` value, but no `bounds` value:

```
dlg.btnPnl = dlg.add('panel', undefined, 'Build it');
dlg.btnPnl.testBtn = dlg.btnPnl.add('button', undefined, 'Test');
```

Tip: This example creates a dynamic property, `btnPnl`, on the parent window object, which contains the returned reference to the child control object. This is not required, but provides a useful way to access your controls.

A new element is initially set to be visible, but it not shown unless its parent object is shown.

Creation properties

Some element types have attributes that can only be specified when the element is created. These are not normal properties of the element, in that they cannot be changed during the element's lifetime, and they are only needed once. For these element types, you can supply an optional *creation-properties* argument to the `add` method. This argument is an object with one or more properties that control aspects of the element's appearance, or special functions such as whether an edit text element is editable or read-only. See Control object constructors for details.

All UI elements have an optional creation property called `name`, which assigns a name for identifying that element. For example, the following creates a new `Button` element with the name 'ok':

```
dlg.btnPnl.buildBtn =
  dlg.btnPnl.add('button', undefined, 'Build', {name:'ok'});
```

Accessing child elements

A reference to each element added to a container is appended to the container's `children` property. You can access the child elements through this array, using a 0-based index. For controls that are not containers, the `children` collection is empty.

In this example, the `msgPnl` panel was the first element created in `dlg`, so the script can access the panel object at index 0 of the parent's `children` property to set the text for the title:

```
var dlg = new Window('dialog', 'Alert Box Builder');
dlg.msgPnl = dlg.add('panel');
dlg.children[0].text = 'Messages';
```

If you use a creation property to assign a name to a newly created element, you can access that child by its name, either in the `children` array of its parent, or directly as a property of its parent. For example, the `Button` in a previous example was named "`ok`", so it can be referenced as follows:

```
dlg.btnPnl.children['ok'].text = "Build";
dlg.btnPnl.ok.text = "Build";
```

For list controls (type `list` and `dropdown`), you can access the child list-item objects through the `items` array.

Removing elements

To add elements to a `window`, `panel`, or `group`, use the container's remove method. This method accepts an object representing the element to be removed, or the name of the element, or the index of the element in the container's `children` collection (see Accessing child elements).

The specified element is removed from view if it was currently visible, and it is no longer accessible from the container or window. The results of any further references by a script to the object representing the element are undefined.

To remove list items from a list, use the parent list control's remove method in the same way. It removes the item from the parent's `items` list, hides it from view, and deletes the item object.

Types of controls

The following sections introduce the types of controls you can add to a `Window` or other container element (`panel` or `group`). For details of the properties and functions, and of how to create each type of element, see ScriptUI Object Reference.

Containers

These are types of Control Objects which are contained in windows, and which contain and group other controls.

<table>
<tr><td><code>Panel</code></td><td>Typically used to visually organize related controls.
● Set the <code>text</code> property to define a title which appears at the top of the <code>Panel</code>.
● An optional <code>borderStyle</code> creation property controls the appearance of the border drawn around the panel.
You can use <code>Panels</code> as separators: those with <code>width = 0</code> appear as vertical lines and those with <code>height = 0</code> appear as horizontal lines.
<pre>var dlg = new Window('dialog', 'Alert Box Builder',[100,100,480,245]);
dlg.msgPnl = dlg.add('panel', [25,15,355,130], 'Messages');</pre></td></tr>
<tr><td><code>Group</code></td><td>Used to visually organize related controls. Unlike <code>Panels</code>, <code>Groups</code> have no title or visible border. You can use them to create hierarchies of controls, and for fine control over layout attributes of certain groups of controls within a larger panel. For examples, see Creating more complex arrangements.</td></tr>
</table>

User interface controls

These are types of Control Objects which are contained in windows, panels, and groups, and which provide specific kinds of display and user interaction.

`StaticText`

Typically used to display text strings that are not intended for direct manipulation by a user, such as informative messages or labels.

This example creates a `Panel` and adds several `StaticText` elements:

```
var dlg = new Window('dialog', 'Alert Box Builder',[100,100,480,245]);
dlg.msgPnl = dlg.add('panel', [25,15,355,130], 'Messages');
dlg.msgPnl.titleSt = dlg.msgPnl.add('statictext', [15,15,105,35],
   'Alert box title:');
dlg.msgPnl.msgSt = dlg.msgPnl.add('statictext', [15,65,105,85],
   'Alert message:');
dlg.show();
```

`EditText`

Allows users to enter text, which is returned to the script when the dialog is dismissed. Text in `EditText` elements can be selected, copied, and pasted.

- Set the `text` property to assign the initial displayed text in the element, and read it to obtain the current text value, as entered or modified by the user.
- Set the `textselection` property to replace the current selection with new text, or to insert text at the cursor (insertion point). Read this property to obtain the current selection, if any.

This example adds some `EditText` elements, with initial values that a user can accept or replace:

```
var dlg = new Window('dialog', 'Alert Box Builder',[100,100,480,245]);
dlg.msgPnl = dlg.add('panel', [25,15,355,130], 'Messages');
dlg.msgPnl.titleSt = dlg.msgPnl.add('statictext', [15,15,105,35],
   'Alert box title:');
dlg.msgPnl.titleEt = dlg.msgPnl.add('edittext', [115,15,315,35],
   'Sample Alert');
dlg.msgPnl.msgSt = dlg.msgPnl.add('statictext', [15,65,105,85],
   'Alert message:');
dlg.msgPnl.msgEt = dlg.msgPnl.add('edittext', [115,45,315,105],
   '<your message here>', {multiline:true});
dlg.show();
```

Note the creation property on the second `EditText` field, where `multiline:true` indicates a field in which a long text string can be entered. The text wraps to appear as multiple lines.

`Button`

Typically used to initiate some action from a window when a user clicks the button; for example, accepting a dialog's current settings, canceling a dialog, bringing up a new dialog, and so on.

- Set the `text` property to assign a label to identify a `Button`'s function.
- The `onClick` callback method provides behavior.

```
var dlg = new Window('dialog', 'Alert Box Builder',[100,100,480,245]);
dlg.btnPnl = dlg.add('panel', [15,50,365,95], 'Build it');
dlg.btnPnl.testBtn = dlg.btnPnl.add('button', [15,15,115,35], 'Test');
dlg.btnPnl.buildBtn = dlg.btnPnl.add('button', [125,15,225,35],
   'Build', {name:'ok'});
dlg.btnPnl.cancelBtn = dlg.btnPnl.add('button', [235,15,335,35],
   'Cancel', {name:'cancel'});
dlg.show();
```

`IconButton`

A button that displays an icon instead of text. Like a text button, typically initiates an action in response to a click.

- The `icon` property identifies the icon image; see Displaying icons.
- The `onClick` callback method provides behavior.

`Image`	Displays an iconic image. ● The `icon` property identifies the icon image; see Displaying icons.

`Checkbox`	Allows the user to set a boolean state.
	● Set the `text` property to assign an identifying text string that appears next to the clickable box.
	● The user can click to select or deselect the box, which shows a checkmark when selected. The `value=true` when it is selected (checked) and `false` when it is not.
	When you create a `Checkbox`, you can set its `value` property to specify its initial state and appearance.
	`// Add a checkbox to control the buttons that dismiss an alert box`
	`dlg.hasBtnsCb = dlg.add('checkbox', [125,145,255,165],`
	`   'Should there be alert buttons?');`
	`dlg.hasBtnsCb.value = true;`

`RadioButton`	Allows the user to select one choice among several.
	● Set the `text` property to assign an identifying text string that appears next to the clickable button.
	● The `value=true` when the button is selected. The button shows the state in a platform-specific manner, with a filled or empty dot, for example.
	You group a related set of radio buttons by creating all the related elements one after another. When any button's value becomes `true`, the value of all other buttons in the group becomes `false`. When you create a group of radio buttons, you should set the state of one of them `true`:
	`var dlg = new Window('dialog', 'Alert Box Builder',[100,100,480,245]);`
	`dlg.alertBtnsPnl = dlg.add('panel', [45,50,335,95], 'Button alignment');`
	`dlg.alertBtnsPnl.alignLeftRb = dlg.alertBtnsPnl.add('radiobutton',`
	`   [15,15,95,35], 'Left');`
	`dlg.alertBtnsPnl.alignCenterRb = dlg.alertBtnsPnl.add('radiobutton',`
	`   [105,15,185,35], 'Center');`
	`dlg.alertBtnsPnl.alignRightRb = dlg.alertBtnsPnl.add('radiobutton',`
	`   [195,15,275,35], 'Right');`
	`dlg.alertBtnsPnl.alignCenterRb.value = true``;`
	`dlg.show();`

`Progressbar`	Typically used to display the progress of a time-consuming operation. A colored bar covers a percentage of the area of the control, representing the percentage completion of the operation. The `value` property reflects and controls how much of the visible area is colored, relative to the maximum value (`maxvalue`). By default the range is 0 to 100, so the `value=50` when the operation is half done.
`Slider`	Typically used to select within a range of values. The slider is a horizontal bar with a draggable indicator, and you can click a point on the slider bar to jump the indicator to that location. The `value` property reflects and controls the position of the indicator, within the range determined by `minvalue` and `maxvalue`. By default the range is 0 to 100, so setting `value=50` moves the indicator to the middle of the bar.

<table>
<tr><td>Scrollbar</td><td>Like a slider, the scrollbar is a bar with a draggable indicator. It also has "stepper" buttons at each end, that you can click to jump the indicator by the amount in the <code>stepdelta</code> property. If you click a point on the bar outside the indicator, the indicator jumps by the amount in the <code>jumpdelta</code> property.

You can create scrollbars with horizontal or vertical orientation; if <code>width</code> is greater than <code>height</code>, it is horizontal, otherwise it is vertical.

Scrollbars are often created with an associated <code>EditText</code> field to display the current value of the scrollbar, and to allow setting the scrollbar's position to a specific value. This example creates a scrollbar with associated <code>StaticText</code> and <code>EditText</code> elements within a panel:
<pre>dlg.sizePnl = dlg.add('panel', [60,240,320,315], 'Dimensions');
dlg.sizePnl.widthSt = dlg.sizePnl.add('statictext', [15,15,65,35],
   'Width:');
dlg.sizePnl.widthScrl = dlg.sizePnl.add('scrollbar', [75,15,195,35],
   300, 300, 800);
dlg.sizePnl.widthEt = dlg.sizePnl.add('edittext', [205,15,245,35]);</pre>The last three arguments to the <code>add</code> method that creates the scrollbar define the values for the <code>value</code>, <code>minvalue</code> and <code>maxvalue</code> properties.</td></tr>
<tr><td>ListBox
DropDownList</td><td>These controls display lists of items, which are represented by <code>ListItem</code> objects in the <code>items</code> property. You can access the items in this array using a 0-based index.
● A <code>ListBox</code> control displays a list of choices. When you create the object, you specify whether it allows the user to select only one or multiple items. If a list contains more items than can be displayed in the available area, a scrollbar may appear that allows the user to scroll through all the list items.
● A <code>DropDownList</code> control displays a single visible item. When you click the control, a list drops down and allows you to select one of the other items in the list. Drop-down lists can have nonselectable separator items for visually separating groups of related items, as in a menu.

You can specify the choice items on creation of the list object, or afterward using the list object's add method. You can remove items programmatically with the list object's remove and removeAll method.</td></tr>
<tr><td>ListItem</td><td>Items added to or inserted into any type of list control are <code>ListItem</code> objects, with properties that can be manipulated from a script. <code>ListItem</code> elements can be of the following types:
● <code>item</code>: the typical item in any type of list. It displays text or an icon, and can be selected. To display an icon, set the item object's <code>icon</code> property; see Displaying icons.
● <code>separator</code>: a separator is a nonselectable visual element in a drop-down list. Although it has a <code>text</code> property, the value is ignored, and the item is displayed as a horizontal line.</td></tr>
</table>

Displaying icons

You can display icon images in Image or IconButton controls, or in place of strings as the selectable items in a Listbox or DropdownList control. In each case, the image is defined by setting the element's `icon` property, either to a named icon resource, a File Object, or the pathname of a file containing the iconic image (see Specifying Paths).

The image data for an icon must be in Portable Network Graphics (PNG) format. See http://www.libpng.org for detailed information on the PNG format.

You can set or reset the `icon` property at any time to change the image displayed in the element.

The scripting environment can define icon *resources*, which are available to scripts by name. To specify an icon resource, set a control's `icon` property to the resource's JavaScript name, or refer to the resource by name when creating the control. For example, to create a button with an application-defined icon resource:

```
myWin.upBtn = myWin.add ("iconbutton", undefined, "SourceFolderIcon");
```

If a script does not explicitly set the `preferredSize` or `size` property of an element that displays a icon image, the value of `preferredSize` is determined by the dimensions of the iconic image. If the size values are explicitly set to dimensions smaller than those of the actual image graphic, the displayed image is clipped. If they are set to dimensions larger than those of the image graphic, the displayed image is centered in the larger space. An image is never scaled to fit the available space.

Prompts and alerts

Static functions on the Window Class are globally available to display short messages in standard dialogs. The host application controls the appearance of these simple dialogs, so they are consistent with other alert and message boxes displayed by the application. You can often use these standard dialogs for simple interactions with your users, rather than designing special-purpose dialogs of your own.

Use the static functions alert, confirm, and prompt on the `Window` class to invoke these dialogs with your own messages. You do not need to create a `window` object to call these functions.

Modal dialogs

A modal dialog is initially invisible. Your script invokes it using the show method, which does not return until the dialog has been dismissed. The user can dismiss it by using a platform-specific window gesture, or by using one of the dialog controls that you supply, typically an **OK** or **Cancel** button. The onClick method of such a button must call the close or hide method to close the dialog. The `close` method allows you to pass a value to be returned by the `show` method.

For an example of how to define such buttons and their behavior, see Defining Behavior for Controls with Event Callbacks.

Creating and using modal dialogs

A dialog typically contains some controls that the user must interact with, to make selections or enter values that your script will use. In some cases, the result of the user action is stored in the object, and you can retrieve it after the dialog has been dismissed. For example, if the user changes the state of a `Checkbox` or `RadioButton`, the new state is found in the control's `value` property.

However, if you need to respond to a user action while the dialog is still active, you must assign the control a callback function for the interaction event, either onClick or onChange. The callback function is the value of the `onClick` or `onChange` property of the control.

For example, if you need to validate a value that the user enters in a `edittext` control, you can do so in an `onChange` callback handler function for that control. The callback can perform the validation, and perhaps display an alert to inform the user of errors.

Sometimes, a modal dialog presents choices to the user that must be correct before your script allows the dialog to be dismissed. If your script needs to validate the state of a dialog after the user clicks **OK**, you can define an onClose event handler for the dialog. This callback function is invoked whenever a window is closed. If the function returns `true`, the window is closed, but if it returns `false`, the close operation is cancelled and the window remains open.

Your `onClose` handler can examine the states of any controls in the dialog to determine their correctness, and can show alert messages or use other modal dialogs to alert the user to any errors that must be corrected. It can then return `true` to allow the dialog to be dismissed, or `false` to allow the user to correct any errors.

Dismissing a modal dialog

Every modal dialog should have at least one button that the user can click to dismiss the dialog. Typically modal dialogs have an **OK** and a **Cancel** button to close the dialog with or without accepting changes that were made in it.

You can define onClick callbacks for the buttons that close the parent dialog by calling its close method. You have the option of sending a value to the `close` method, which is in turn passed on to and returned from the show method that invoked the dialog. This return value allows your script to distinguish different closing events; for example, clicking **OK** can return 1, clicking **Cancel** can return 2. However, for this typical behavior, you do not need to define these callbacks explicitly; see Default and cancel elements below.

For some dialogs, such as a simple alert with only an **OK** button, you do not need to return any value. For more complex dialogs with several possible user actions, you might need to distinguish more outcomes. If you need to distinguish more than two closing states, you must define your own closing callbacks rather than relying on the default behavior.

If, by mistake, you create a modal dialog with no buttons to dismiss it, or if your dialog does have buttons, but their `onClick` handlers do not function properly, a user can still dismiss the dialog by typing ESC. In this case, the system will execute a call to the dialog's `close` method, passing a value of 2. This is not, of course, a recommended way to design your dialogs, but is provided as an escape hatch to prevent the application from hanging in case of an error in the operations of your dialog.

Default and cancel elements

The user can typically dismiss a modal dialog by clicking an **OK** or **Cancel** button, or by typing certain keyboard shortcuts. By convention, typing ENTER is the same as clicking **OK** or the default button, and typing ESC is the same as clicking **Cancel**. The keyboard shortcut has the same effect as calling notify for the associated `button` control.

To determine which control is notified by which keyboard shortcut, set the `dialog` object's defaultElement and cancelElement properties. The value is the control object that should be notified when the user types the associated keyboard shortcut.

- For buttons assigned as the `defaultElement`, if there is no `onClick` handler associated with the button, clicking the button or typing ENTER calls the parent dialog's close method, passing a value of 1 to be returned by the show call that opened the dialog.
- For buttons assigned as the `cancelElement`, if there is no `onClick` handler associated with the button, clicking the button or typing ESC calls the parent dialog's close method, passing a value of 2 to be returned by the show call that opened the dialog.

If you do not set the `defaultElement` and `cancelElement` properties explicitly, ScriptUI tries to choose reasonable defaults when the dialog is about to be shown for the first time. For the default element, it looks for a button whose `name` or `text` value is "`ok`" (disregarding case). For the cancel element, it looks for a button whose `name` or `text` value is "`cancel`" (disregarding case). Because it looks at the `name` value first, this works even if the `text` value is localized. If there is no suitable button in the dialog, the property value remains `null`, which means that the keyboard shortcut has no effect in that dialog.

To make this feature most useful, it is recommended that you always provide the `name` creation property for buttons meant to be used in this way.

Resource Specifications

You can create one or more UI elements at a time using a *resource specification*. This specially formatted string provides a simple and compact means of creating an element, including any container element and its component elements. The resource-specification string is passed as the `type` parameter to the `Window()` or `add()` constructor function.

The general structure of a resource specification is an element type specification (such as `dialog`), followed by a set of braces enclosing one or more property definitions.

```
var myResource = "dialog{ control_specs }";
var myDialog = new Window ( myResource );
```

Controls are defined as properties within windows and other containers. For each control, give the class name of the control, followed by the properties of the control enclosed in braces. For example, the following specifies a `Button`:

```
testBtn: Button { text: 'Test' }
```

The following resource string specifies a panel that contains several `StaticText` and `EditText` controls:

```
"msgPnl: Panel { text: 'Messages', bounds:[25,15,355,130], \
  titleSt: StaticText { text:'Alert box title:', \
     bounds:[15,15,105,35] }, \
  titleEt: EditText { text:'Sample Alert', bounds:[115,15,315,35] }, \
  msgSt: StaticText { text:'Alert message:', \
     bounds:[15,65,105,85] }, \
  msgEt: EditText { text:'<your message here>', \
     bounds:[115,45,315,105], properties:{multiline:true} } \
}"
```

The property with name `properties` specifies creation properties; see Creation properties.

A property value can be specified as `null`, `true`, `false`, a string, a number, an inline array, or an object.

- An inline array contains one or more values in the form:

  `[value, value,...]`

- An object can be an inline object, or a named object, in the form:

  `{classname inlineObject}`

- An inline object contains one or more properties, in the form:

  `{propertyName:propertyValue,propertyName:propertyValue,... }`

The Resource specification example shows how to build a complete window and all its contents with a resource specification. The resource specification format can also be used to create a single element or container and its child elements. For example, if the `alertBuilderResource` in Resource specification example did not contain the panel `btnPnlResource`, you could define that resource separately, then add it to the dialog as follows:

```
var btnPnlResource =
   "panel { text: 'Build it', bounds:[15,330,365,375], \
      testBtn: Button { text:'Test', bounds:[15,15,115,35] }, \
      buildBtn: Button { text:'Build', bounds:[125,15,225,35], \
      properties:{name:'ok'} }, \
      cancelBtn: Button { text:'Cancel', bounds:[235,15,335,35], \
      properties:{name:'cancel'} } \
   }";
dlg = new Window(alertBuilderResource);
dlg.btnPnl = dlg.add(btnPnlResource);
dlg.show();
```

Defining Behavior for Controls with Event Callbacks

You must define the behavior of your controls in order for them to respond to user interaction. You do this by defining event-handling callback functions as part of the definition of the control or window. To respond to a specific event, define a handler function for it, and assign a reference to that function in the corresponding property of the window or control object. Different types of windows and controls respond to different actions, or events:

- Windows generate events when the user moves or resizes the window. To handle these events, define callback functions for onMove, onMoving, onResize, and onResizing. To respond to the user opening or closing the window, define callback functions for onShow and onClose.
- Button, radiobutton, and checkbox controls generate events when the user clicks within the control bounds. To handle the event, define a callback function for onClick.
- Edittext, scrollbar, and slider controls generate events when the content or value changes—that is, when the user types into an edit field, or moves the scroll or slider indicator. To handle these events, define callback functions for onChange and onChanging.

Defining event handler functions

Your script can define an event handler as a named function referenced by the callback property, or as an unnamed function defined inline in the callback property.

- If you define a named function, assign its name as the value of the corresponding callback property. For example:

```
function hasBtnsCbOnClick { /* do something interesting */ }
hasBtnsCb.onClick = hasBtnsCbOnClick;
```

- For a simple, unnamed function, set the property value directly to the function definition:

```
UI_element.callback_name = function () { handler_definition};
```

Event-handler functions take no arguments.

For example, the following sets the `onClick` property of the checkbox `hasBtnsCb`, to a function that enables another control in the same dialog:

```
hasBtnsCb.onClick = function ()
   { this.parent.alertBtnsPnl.enabled = this.value; };
```

The following statements set the `onClick` event handlers for buttons that close the containing dialog, returning different values to the `show` method that invoked the dialog, so that the calling script can tell which button was clicked:

```
buildBtn.onClick = function () { this.parent.parent.close(1); };
cancelBtn.onClick = function () { this.parent.parent.close(2); };
```

Simulating user events

You can simulate user actions by sending an event notification directly to a window or control with the `notify` method. A script can use this method to generate events in the controls of a window, as if a user was clicking buttons, entering text, or moving the window. If you have defined an event-handler callback for the element, the `notify` method invokes it.

The `notify` method takes an optional argument that specifies which event it should simulate. If a control can generate only one kind of event, notification generates that event by default.

The following controls generate the `onClick` event:

```
button
checkbox
iconbutton
radiobutton
```

The following controls generate the `onChange` event:

```
dropdownlist
edittext
listbox
scrollbar
slider
```

The following controls generate the `onChanging` event:

```
edittext
scrollbar
slider
```

In `radiobutton` and `checkbox` controls, the boolean `value` property automatically changes when the user clicks the control. If you use `notify()` to simulate a click, the `value` changes just as if the user had clicked. For example, if the `value` of a checkbox `hasBtnsCb` is `true`, this code changes the value to `false`:

```
if (dlg.hasBtnsCb.value == true) dlg.hasBtnsCb.notify();
// dlg.hasBtnsCb.value is now false
```

Automatic Layout

When a script creates a window and its associated UI elements, it can explicitly control the size and location of each element and of the container elements, or it can take advantage of the automatic layout capability provided by ScriptUI. The automatic layout mechanism uses certain available information about UI elements, along with a set of layout rules, to establish a visually pleasing layout of the controls in a dialog, automatically determining the proper sizes for elements and containers.

Automatic layout is easier to program than explicit layout. It makes a script easier to modify and maintain, and it also makes the script easier to localize for different languages.

The script programmer has considerable control over the automatic layout process. Each container has an associated layout manager object, specified in the `layout` property. The layout manager controls the sizes and positions of the contained elements, and also sizes the container itself.

There is a default layout manager object, or you can create a new one:

```
myWin.layout = new AutoLayoutManager(myWin);
```

Default layout behavior

By default, the `autoLayoutManager` object implements the default layout behavior. A script can modify the properties of the default layout manager object, or create a new, custom layout manager if it needs more specialized layout behavior. See Custom layout manager example.

Child elements of a container can be organized in a single row or column, or in a stack, where the elements overlap one other in the same region of the container, and only the top element is fully visible. This is controlled by the container's `orientation` property, which can have the value `row`, `column`, or `stack`.

You can nest `Panel` and `Group` containers to create more complex organizations. For example, to display two columns of controls, you can create a panel with a row orientation that in turn contains two groups, each with a column orientation.

Containers have properties to control inter-element spacing and margins within their edges. The layout manager provides defaults if these are not set.

The alignment of child elements within a container is controlled by the `alignChildren` property of the container, and the `alignment` property of the individual controls. The `alignChildren` property determines an overall strategy for the container, which can be overridden by a particular child element's `alignment` value.

A layout manager can determine the best size for a child element through the element's `preferredSize` property. The value defaults to dimensions determined by the UI framework based on characteristics of the control type and variable characteristics such as a displayed text string.

For details of how you can set these property values to affect the automatic layout, see Automatic layout properties.

Note: ScriptUI does not offer direct control of fonts, and fonts are chosen differently on different platforms, so windows that are created the same way can appear different on different platforms.

Automatic layout properties

Your script establishes rules for the layout manager by setting the values of certain properties, both in the container object and in the child elements. The following examples show the effects of various combinations of values for these properties. The examples are based on a simple window containing a `StaticText`, `Button` and `EditText` element, created (using Resource Specifications) as follows:

```
var w = new Window(
   "window { \
      orientation: 'row', \
      st: StaticText { }, \
      pb: Button { text: 'OK' }, \
      et: EditText { size:[20, 30] } \
   }");
w.show();
```

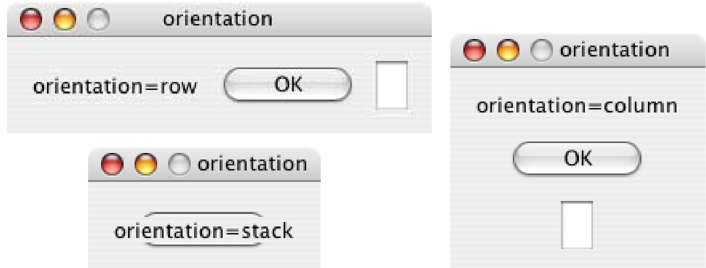
Each example shows the effects of setting particular layout properties in various ways. In each window, `w.text` is set so that the window title shows which property is being varied, and `w.st.text` is set to display the particular property value being demonstrated.

Container orientation

The `orientation` property of a container specifies the organization of child elements within it. It can have these values:

- `row`: Child elements are arranged next to each other, in a single row from left to right across the container. The height of the container is based on the height of the tallest child element in the row, and the width of the container is based on the combined widths of all the child elements.
- `column`: Child elements are arranged above and below each other, in a single column from top to bottom across the container. The height of the container is based on the combined heights of all the child elements, and the width of the container is based on the widest child element in the column.
- `stack`: Child elements are arranged overlapping one another, as in a stack of papers. The elements overlie one another in the same region of the container. Only the top element is fully visible. The height of the container is based on the height of the tallest child element in the stack, and the width of the container is based on the widest child element in the stack.

The following figure shows the results of laying out the sample window with each of these orientations:



Aligning children

The alignment of child elements within a container is controlled by two properties: `alignChildren` in the parent container, and `alignment` in each child. The `alignChildren` value in the parent container controls the alignment of all children within that container, unless it is overridden by the `alignment` value set on an individual child element.

These properties use the same values, which specify alignment along one axis, depending on the orientation of the container. The property values are not case-sensitive; for example, the strings `FILL`, `Fill`, and `fill` are all valid.

Elements in a row can be aligned along the vertical axis, in these ways:

- `top`: The element's top edge is located at the top margin of its container.
- `bottom`: The element's bottom edge is located at the bottom margin of its container.
- `center`: The element is centered within the top and bottom margins of its container.
- `fill`: The element's height is adjusted to fill the height of the container between the top and bottom margins.

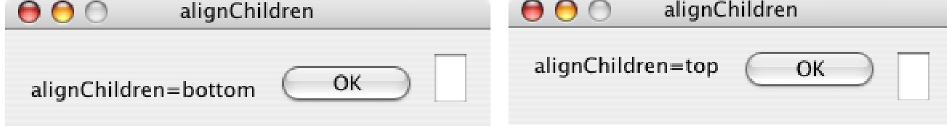
Elements in a column can be aligned along the horizontal axis, in these ways:

- `left`: The element's left edge is located at the left margin of its container.
- `right`: The element's right edge is located at the right margin of its container.
- `center`: The element is centered within the right and left margins of its container.
- `fill`: The element's width is adjusted to fill the width of the container between the right and left margins.

Elements in a stack can be aligned along either the vertical or the horizontal axis, in these ways:

- `top`: The element's top edge is located at the top margin of its container, and the element is centered within the right and left margins of its container.
- `bottom`: The element's bottom edge is located at the bottom margin of its container, and the element is centered within the right and left margins of its container.
- `left`: The element's left edge is located at the left margin of its container, and the element is centered within the top and bottom margins of its container.
- `right`: The element's right edge is located at the right margin of its container, and the element is centered within the top and bottom margins of its container.
- `center`: The element is centered within the top, bottom, right and left margins of its container.
- `fill`: The element's height is adjusted to fill the height of the container between the top and bottom margins., and the element's width is adjusted to fill the width of the container between the right and left margins.

The following figure shows the results of creating the sample window with row orientation and the `bottom` and `top` alignment settings in the parent's `alignChildren` property:



The following figure shows the results of creating the sample window with column orientation and the `right`, `left`, and `fill` alignment settings in the parent's `alignChildren` property. Notice how in the `fill` case, each element is made as wide as the widest element in the container:

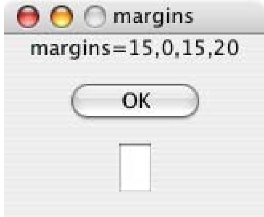
You can override the container's child alignment, as specified by `alignChildren`, by setting the `alignment` property of a particular child element. The following diagram shows the result of setting `alignment` to `right` for the `EditText` element, when the parent's `alignChildren` value is left:

Setting margins

The `margins` property of a container specifies the number of pixels between the edges of a container and the outermost edges of the child elements. You can set this property to a simple number to specify equal margins, or using a Margins object, which allows you to specify different margins for each edge of the container.

The following figure shows the results of creating the sample window with row orientation and margins of 5 and 15 pixels:

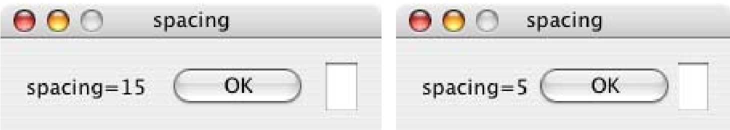
This figure shows the results of creating the sample window with column orientation, a top margin of 0 pixels, a bottom margin of 20 pixels, and left and right margins of 15 pixels:



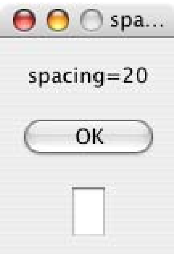
Spacing between children

The `spacing` property of a container specifies the number of pixels separating one child element from its adjacent sibling element.

This figure shows the results of creating the sample window with row orientation, and spacing of 15 and 5 pixels, respectively:



This figure shows the results of creating the sample window with column orientation, and spacing of 20 pixels:



Determining a preferred size

Each element has a `preferredSize` property, which is initially defined with reasonable default dimensions for the element. The default value is calculated by ScriptUI, and is based on constant characteristics of each type of element, and variable characteristics such as the text string to be displayed in a button or text element.

If an element's `size` property is not defined, the layout manager uses the value of `preferredSize` to determine the dimensions of each element during the layout process. Generally, you should avoid setting the `preferredSize` property explicitly, and let ScriptUI determine the best value based on the state of an element at layout time. This allows you to set the `text` properties of your UI elements using localizable strings (see Localization in ScriptUI Objects). The width and height of each element are calculated at layout time based on the chosen language-specific text string, rather than relying on the script to specify a fixed size for each element.

However, a script can explicitly set the `preferredSize` property to give hints to the layout manager about the intended sizes of elements for which a reasonable default size is not easily determined, such as an `EditText` element that has no initial text content to measure.

Creating more complex arrangements

You can easily create more complex arrangements by nesting `Group` containers within `Panel` containers and other `Group` containers.

Many dialogs consist of rows of information to be filled in, where each row has columns of related types of controls. For instance, an edit field is typically in a row next to a static text label that identifies it, and a series of such rows are arranged in a column. This example (created using Resource Specifications) shows a simple dialog in which a user can enter information into two `EditText` fields, each arranged in a row

with its `StaticText` label. To create the layout, a `Panel` with a column orientation contains two `Group` elements with row orientation. These groups contain the control rows. A third `Group`, outside the panel, contains the row of buttons.

```
res =
"dialog { \
   info: Panel { orientation: 'column', \
      text: 'Personal Info', \
      name: Group { orientation: 'row', \
         s: StaticText { text:'Name:' }, \
         e: EditText { preferredSize: [200, 20] } \
      }, \
      addr: Group { orientation: 'row', \
         s: StaticText { text:'Street / City:' }, \
         e: EditText { preferredSize: [200, 20] } \
      }  \
   }, \
   buttons: Group { orientation: 'row', \
      okBtn: Button { text:'OK', properties:{name:'ok'} }, \
      cancelBtn: Button { text:'Cancel', properties:{name:'cancel'} } \
   } \
}";
win = new Window (res);
win.center();
win.show();
```

In this simplest example, the columns are not vertically aligned. When you are using fixed-width controls in your rows, a simple way to get an attractive alignment of the `StaticText` labels for your `EditText` fields is to align the child rows in the `Panel` to the right of the panel. In the example, add the following to the `Panel` specification:

```
info: Panel { orientation: 'column', alignChildren:'right', \
```

This creates the following result:

Suppose now that you need two panels, and want each panel to have the same width in the dialog. You can specify this at the level of the dialog window object, the parent of both panels. Specify `alignChildren='fill'`, which makes each child of the dialog match its width to the widest child.

```
res =
   "dialog { alignChildren: 'fill', \
      info: Panel { orientation: 'column', alignChildren:'right', \
```

```
            text: 'Personal Info', \
            name: Group { orientation: 'row', \
               s: StaticText { text:'Name:' }, \
               e: EditText { preferredSize: [200, 20] } \
            } \
         }, \
         workInfo: Panel { orientation: 'column', \
            text: 'Work Info', \
            name: Group { orientation: 'row', \
               s: StaticText { text:'Company name:' }, \
               e: EditText { preferredSize: [200, 20] } \
            } \
         }, \
         buttons: Group { orientation: 'row', alignment: 'right', \
            okBtn: Button { text:'OK', properties:{name:'ok'} }, \
            cancelBtn: Button { text:'Cancel', properties:{name:'cancel'} } \
         } \
      }";
   win = new Window (res); win.center(); win.show();
```

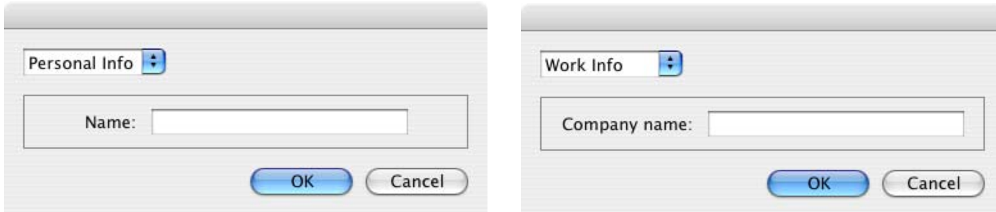
To make the buttons to appear at the right of the dialog, the `buttons` group overrides the `fill` alignment of its parent (the dialog), and specifies `alignment='right'`.

Creating dynamic content

Many dialogs need to present different sets of information based on the user selecting some option within the dialog. You can use the `stack` orientation to present different views in the same region of a dialog.

A `stack` orientation of a container places child elements so they are centered in a space which is wide enough to hold the widest child element, and tall enough to contain the tallest child element. If you arrange groups or panels in such a stack, you can show and hide them in different combinations to display a different set of controls in the same space, depending on other choices in the dialog.

For example, this dialog changes dynamically according to the user’s choice in the `DropDownList`.



The following script creates this dialog. It compresses the "Personal Info" and "Work Info" panels from the previous example into a single `Panel` that has two `Groups` arranged in a stack. A `DropDownList` allows the user to choose which set of information to view. When the user makes a choice in the list, its onChange function shows one group, and hides the other.

```
res =
   "dialog { \
      whichInfo: DropDownList { alignment:'left' }, \
      allGroups: Panel { orientation:'stack', \
         info: Group { orientation: 'column', \
            name: Group { orientation: 'row', \
               s: StaticText { text:'Name:' }, \
               e: EditText { preferredSize: [200, 20] } \
            } \
         }, \
         workInfo: Group { orientation: 'column', \
            name: Group { orientation: 'row', \
               s: StaticText { text:'Company name:' }, \
               e: EditText { preferredSize: [200, 20] } \
            } \
            }, \
      }, \
      buttons: Group { orientation: 'row', alignment: 'right', \
         okBtn: Button { text:'OK', properties:{name:'ok'} }, \
         cancelBtn: Button { text:'Cancel', properties:{name:'cancel'} } \
      } \
   }";
win = new Window (res);
win.whichInfo.onChange = function () {
   if (this.selection != null) {
      for (var g = 0; g < this.items.length; g++)
         this.items[g].group.visible = false; //hide all other groups
      this.selection.group.visible = true;//show this group
   }
}
var item = win.whichInfo.add ('item', 'Personal Info');
item.group = win.allGroups.info;
item = win.whichInfo.add ('item', 'Work Info');
item.group = win.allGroups.workInfo;
win.whichInfo.selection = win.whichInfo.items[0];
win.center();
win.show();
```

Custom layout manager example

This script creates a dialog almost identical to the one in the previous example, except that it defines a layout-manager subclass, and assigns an instance of this class as the `layout` property for the last `Group` in the dialog. (The example also demonstrates the technique for defining a reusable class in JavaScript.)

This script-defined layout manager positions elements in its container in a stair-step fashion, so that the buttons are staggered rather than in a straight line.

```
/* Define a custom layout manager that arranges the children
** of 'container' in a stair-step fashion.*/

function StairStepButtonLayout (container) { this.initSelf(container); }

// Define its 'method' functions
function SSBL_initSelf (container) { this.container = container; }

function SSBL_layout() {
  var top = 0, left = 0;
  var width;
  var vspacing = 10, hspacing = 20;
  for (i = 0; i < this.container.children.length; i++) {
    var child = this.container.children[i];
    if (typeof child.layout != "undefined")
      // If child is a container, call its layout method
      child.layout.layout();
    child.size = child.preferredSize;
    child.location = [left, top];
    width = left + child.size.width;
    top += child.size.height + vspacing;
    left += hspacing;
  }
  this.container.preferredSize = [width, top - vspacing];
}

// Attach methods to Object's prototype
StairStepButtonLayout.prototype.initSelf = SSBL_initSelf;
StairStepButtonLayout.prototype.layout = SSBL_layout;

// Define a string containing the resource specification for the controls
res =
 "dialog { \
  whichInfo: DropDownList { alignment:'left' }, \
  allGroups: Panel { orientation:'stack', \
    info: Group { orientation: 'column', \
      name: Group { orientation: 'row', \
        s: StaticText { text:'Name:' }, \
        e: EditText { preferredSize: [200, 20] } \
      } \
    }, \
    workInfo: Group { orientation: 'column', \
      name: Group { orientation: 'row', \
        s: StaticText { text:'Company name:' }, \
        e: EditText { preferredSize: [200, 20] } \
```

```
          } \
        }, \
      }, \
      buttons: Group { orientation: 'row', alignment: 'right', \
        okBtn: Button { text:'OK', properties:{name:'ok'} }, \
        cancelBtn: Button { text:'Cancel', properties:{name:'cancel'} } \
      } \
    }";
// Create window using resource spec
win = new Window (res);
// Create list items, select first one
win.whichInfo.onChange = function () {
  if (this.selection != null) {
    for (var g = 0; g < this.items.length; g++)
      this.items[g].group.visible = false;
    this.selection.group.visible = true;
  }
}
var item = win.whichInfo.add ('item', 'Personal Info');
item.group = win.allGroups.info;
item = win.whichInfo.add ('item', 'Work Info');
item.group = win.allGroups.workInfo;
win.whichInfo.selection = win.whichInfo.items[0];

// Override the default layout manager for the 'buttons' group
// with custom layout manager
win.buttons.layout = new StairStepButtonLayout(win.buttons);

win.center();
win.show();
```

The AutoLayoutManager algorithm

When a script creates a `window` object and its elements and shows it the first time, the visible UI-platform window and controls are created. At this point, if no explicit placement of controls was specified by the script, all the controls are located at [0, 0] within their containers, and have default dimensions. Before the window is made visible, the layout manager's `layout` method is called to assign locations and sizes for all the elements and their containers.

The default `AutoLayoutManager`'s `layout` method performs these steps when invoked during the initial call to a `window` object's `show` method:

1. Read the `bounds` property for the managed container; if undefined, proceed with auto layout. If defined, assume that the script has explicitly placed the elements in this container, and cancel the layout operation (if both the `location` and `size` property have been set, this is equivalent to setting the `bounds` property, and layout does not proceed).

2. Determine the container's margins and inter-element spacing from its `margins` and `spacing` properties, and the orientation and alignment of its child elements from the container's `orientation` and `alignChildren` properties. If any of these properties are undefined, use default settings obtained from platform and UI framework-specific default values.

3. Enumerate the child elements, and for each child:

- If the child is a container, call its layout manager (that is, execute this entire algorithm again for the container).
- Read its `alignment` property; if defined, override the default alignment established by the parent container with its `alignChildren` property.
- Read its `size` property: if defined, use it to determine the child’s dimensions. If undefined, read its `preferredSize` property to get the child’s dimensions. Ignore the child's `location` property.

All the per-child information is collected for later use.

4. Based on the orientation, calculate the trial location of each child in the row or column, using inter-element spacing and the container’s margins.
5. Determine the column, row, or stack dimensions, based on the dimensions of the children.
6. Using the desired alignment for each child element, adjust its trial location relative to the edges of its container. For stack orientation, center each child horizontally and vertically in its container.
7. Set the `bounds` property for each child element.
8. Set the container’s `preferredSize` property, based on the margins and dimensions of the row or column of child elements.

Automatic layout restrictions

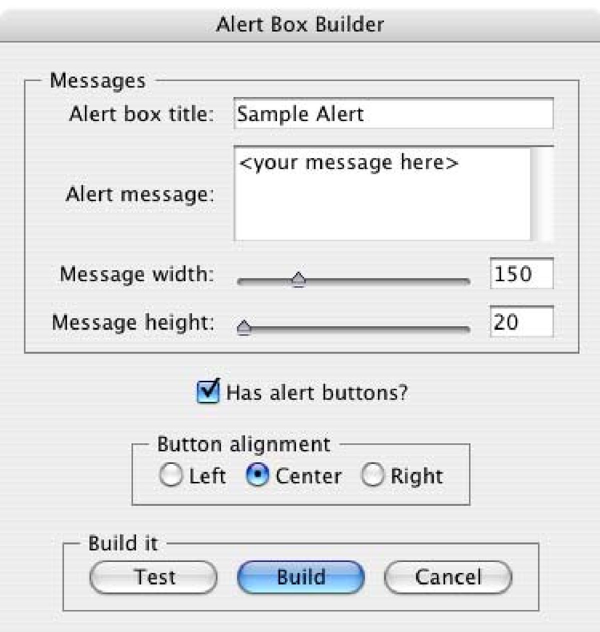
The following restrictions apply to the automatic layout mechanism:

- The default layout manager does not attempt to lay out a container that has a defined `bounds` property. The script programmer can override this behavior by defining a custom layout manager for the container.
- The layout mechanism does not track changes to element sizes after the initial layout has occurred. The script can initiate another layout by calling the layout manager’s `layout` method, and can force the manager to recalculate the sizes of all child containers by passing the optional argument as `true`.
- The layout mechanism does not support re-layout if a dialog window is resized.

Example scripts

These examples demonstrate two ways of building and populating a ScriptUI dialog. The first creates each control with a separate `add` method, while the second defines a resource string that creates the control hierarchy.

The two examples create the same dialog, which collects values from the user. When the Alert Box Builder dialog is dismissed, the script builds a resource string from the collected values, and saves it to a file. That resource string can later be used to create and display the user-configured alert box.



Alert box builder

This variation builds the dialog using the window and panel `add` methods to create each control.

```
//------------- Functions -------------//
/* This function creates the builder dialog using the add method
** An alternative that uses a resource specification is shown
** in the following section */

function createBuilderDialog() {
   // Create an empty dialog window near the upper left of the screen
   var dlg = new Window('dialog', 'Alert Box Builder');
   dlg.frameLocation = [100, 100];
   // Add a panel to hold title and 'message text' strings
   dlg.msgPnl = dlg.add('panel', undefined, 'Messages');
   dlg.msgPnl.alignChildren = "right";
   dlg.msgPnl.title = dlg.msgPnl.add('group');
   dlg.msgPnl.msg = dlg.msgPnl.add('group');
   dlg.msgPnl.msgWidth = dlg.msgPnl.add('group');
   dlg.msgPnl.msgHeight = dlg.msgPnl.add('group');
   with (dlg.msgPnl) {
      title.st = title.add('statictext', undefined, 'Alert box title:');
      title.et = title.add('edittext', undefined, 'Sample Alert');
      title.et.preferredSize = [200,20];
```

```
      msg.st = msg.add('statictext', undefined, 'Alert message:');
      msg.et = msg.add('edittext', undefined, '<your message here>',
        {multiline:true});
      msg.et.preferredSize = [200,60];
      msgWidth.st = msgWidth.add('statictext', undefined, 'Message width:');
      msgWidth.sl = msgWidth.add('slider', undefined, 150, 100, 300);
      msgWidth.sl.preferredSize = [150, 20];
      msgWidth.et = msgWidth.add('edittext');
      msgWidth.et.preferredSize = [40, 20];
      msgHeight.st = msgHeight.add('statictext', undefined, 'Message height:');
      msgHeight.sl = msgHeight.add('slider', undefined, 20, 20, 300);
      msgHeight.sl.preferredSize = [150, 20];
      msgHeight.et = msgHeight.add('edittext');
      msgHeight.et.preferredSize = [40, 20];
    }
    // Add a checkbox to control the presence of buttons to dismiss the alert box
    dlg.hasBtnsCb = dlg.add('checkbox', undefined, 'Has alert buttons?');
    // Add panel to determine alignment of buttons on the alert box
    dlg.alertBtnsPnl = dlg.add('panel', undefined, 'Button alignment');
    dlg.alertBtnsPnl.orientation = "row";
    dlg.alertBtnsPnl.alignLeftRb =
      dlg.alertBtnsPnl.add('radiobutton', undefined, 'Left');
    dlg.alertBtnsPnl.alignCenterRb =
      dlg.alertBtnsPnl.add('radiobutton', undefined, 'Center');
    dlg.alertBtnsPnl.alignRightRb =
      dlg.alertBtnsPnl.add('radiobutton', undefined, 'Right');
    // Add a panel with buttons to test parameters and
    // create the alert box specification
    dlg.btnPnl = dlg.add('panel', undefined, 'Build it');
    dlg.btnPnl.orientation = "row";
    dlg.btnPnl.testBtn = dlg.btnPnl.add('button', undefined, 'Test');
    dlg.btnPnl.buildBtn = dlg.btnPnl.add('button', undefined, 'Build',
      {name:'ok'});
    dlg.btnPnl.cancelBtn =
      dlg.btnPnl.add('button', undefined, 'Cancel', {name:'cancel'});

    return dlg;
  } // createBuilderDialog

  /* This function initializes the values in the controls
  ** of the builder dialog */

  function initializeBuilder(builder) {
    // Set up initial control states
    with (builder) {
      hasBtnsCb.value = true;
      alertBtnsPnl.alignCenterRb.value = true;
      with (msgPnl) {
        msgWidth.et.text = msgWidth.sl.value;
        msgHeight.et.text = msgHeight.sl.value;
      }
    }
    // Attach event callback functions to controls
    /* The 'has buttons' checkbox enables or disables the panel that
       determines the justification of the 'alert' button group */
```

```
    builder.hasBtnsCb.onClick =
      function () { this.parent.alertBtnsPnl.enabled = this.value; };
    /* The edittext fields and scrollbars in msgPnl are connected */
    with (builder.msgPnl) {
      msgWidth.et.onChange =
        function () { this.parent.parent.msgWidth.sl.value = this.text; };
      msgWidth.sl.onChanging =
        function () { this.parent.parent.msgWidth.et.text = this.value; };
      msgHeight.et.onChange =
        function () { this.parent.parent.msgHeight.sl.value = this.text; };
      msgHeight.sl.onChanging =
        function () { this.parent.parent.msgHeight.et.text = this.value; };
    }
    with (builder.btnPnl) {
      // The Test button creates a trial Alert box from
      // the current specifications
      testBtn.onClick =
        function () {
          Window.alert('Type Enter or Esc to dismiss the test Alert box');
          createTestDialog(createResource(this.parent.parent));
        };
      // The Build and Cancel buttons close this dialog
      buildBtn.onClick =
        function () { this.parent.parent.close(1); };
      cancelBtn.onClick =
        function () { this.parent.parent.close(2); };
    };
  } // initializeBuilder

  /* This function invokes the dialog an returns its result */
  function runBuilder(builder) {
    return builder.show();
  }
  /* This function creates and returns a string containing a dialog
  ** resource specification that will create an Alert dialog using
  ** the parameters the user entered in the builder dialog. */
  function createResource(builder) {
    // Define the initial part of the resource spec with dialog parameters
    var res = "dialog { " +
      stringProperty("text", builder.msgPnl.title.et.text) +
      "\n";
    // Define the alert message statictext element, sizing it as user specified
    var textWidth = Number(builder.msgPnl.msgWidth.et.text);
    var textHeight = Number(builder.msgPnl.msgHeight.et.text);
    res += "  msg: StaticText { " +
      stringProperty("text", builder.msgPnl.msg.et.text) +
      " preferredSize: [" + textWidth + ", " + textHeight + "],\n" +
      "    justify:'center', properties:{multiline:true} }";
    // Define buttons if desired
    var hasButtons = builder.hasBtnsCb.value;
    if (hasButtons) {
      var groupAlign = "center";
      // Align buttons as specified
      if (builder.alertBtnsPnl.alignLeftRb.value)
        groupAlign = "left";
```

```
      else if (builder.alertBtnsPnl.alignRightRb.value)
         groupAlign = "right";
      res += ",\n" +
         "  btnGroup: Group {\n" +
         stringProperty("    alignment", groupAlign) +
         "\n" +
         "    okBtn: Button { " +
         stringProperty("text", "OK") +"},\n";
      res += "    cancelBtn: Button { " +
         stringProperty("text", "Cancel") +"}" +
         "  }";
   }
   // done
   res += "\n}";
   return res;
}
function stringProperty(pname, pval) {
   return pname + ":'" + pval + "', ";
}
function createTestDialog(resource) {
   var target = new Window (resource);
   target.center();
   return target.show();
}
//------------- Main script -------------//
var builder = createBuilderDialog(); //for an alternative, see below
initializeBuilder(builder);
if (runBuilder(builder) == 1) {
   // Create the Alert dialog resource specification string
   var resSpec = createResource(builder);
   // Write the resource spec string to a file w/platform file-open dialog
   var fname = File.openDialog('Save resource specification');
   var f = File(fname);
   if (f.open('w')) {
      var ok = f.write(resSpec);
      if (ok)
         ok = f.close();
      if (! ok)
         Window.alert("Error creating " + fname + ": " + f.error);
   }
}
```

Resource specification example

This example provides an alternative method of building the same initial Alert box builder dialog, using a resource specification instead of explicit calls to the `add` method of a container element. To use this alternate version, add this code to the beginning of the previous example in place of the `createBuilderDialog` function. In the main script, replace the line:

```
var builder = createBuilderDialog();
```

with this line:

```
var builder = createBuilderDialogFromResource();
```

The new code follows:

```
var alertBuilderResource =
  "dialog { \
    text: 'Alert Box Builder', frameLocation:[100,100], \
    msgPnl: Panel { orientation:'column', alignChildren:'right', \
      text: 'Messages', \
      title: Group { \
        st: StaticText { text:'Alert box title:' }, \
        et: EditText { text:'Sample Alert', \
            preferredSize:[200, 20] } \
      }, \
      msg: Group { \
        st: StaticText { text:'Alert message:' }, \
        et: EditText { text:'<your message here>', \
            preferredSize:[200, 60], properties:{multiline:true} } \
      }, \
      msgWidth: Group { alignChildren:'center', \
        st: StaticText { text:'Message width:' }, \
        sl: Slider { minvalue:100, maxvalue:300, value:150, \
            preferredSize:[150, 20] }, \
        et: EditText { preferredSize:[40, 20] } \
      }, \
      msgHeight: Group { alignChildren:'center', \
        st: StaticText { text:'Message height:' }, \
        sl: Slider { minvalue:20, maxvalue:300, \
            preferredSize:[150, 20] }, \
        et: EditText { preferredSize:[40, 20] } \
      } \
    }, \
    hasBtnsCb: Checkbox { text:'Has alert buttons?', \
              alignment:'center' }, \
    alertBtnsPnl: Panel { orientation:'row', \
      text: 'Button alignment', \
      alignLeftRb: RadioButton { text:'Left' }, \
      alignCenterRb: RadioButton { text:'Center' }, \
      alignRightRb: RadioButton { text:'Right' } \
    }, \
    btnPnl: Panel { orientation:'row', \
      text: 'Build it', \
      testBtn: Button { text:'Test' }, \
      buildBtn: Button { text:'Build', properties:{name:'ok'} }, \
      cancelBtn: Button { text:'Cancel', properties:{name:'cancel'} } \
    } \
  }";

// This function creates the builder dialog from the resource string
function createBuilderDialogFromResource() {
  var builder = new Window(alertBuilderResource);
  return builder;
} // createBuilderDialogFromResource
```

Localization in ScriptUI Objects

For portions of your user interface that are displayed on the screen, you may want to localize the displayed text. You can localize the display strings in any ScriptUI object (including MenuElement Objects) simply and efficiently, using the Global localize function. This function takes as its argument a *localization object* containing the localized versions of a string.

For complete details of this ExtendScript feature, see Localizing ExtendScript Strings.

A localization object is a JavaScript object literal whose property names are locale names, and whose property values are the localized text strings. The locale name is an identifier as specified in the ISO 3166 standard. In this example, a `btnText` object contains localized text strings for several locales. This object supplies the text for a `Button` to be added to a window `w`:

```
btnText = { en: "Yes", de: "Ja", fr: "Oui" };
b1 = w.add ("button", undefined, localize (btnText));
```

The `localize` function extracts the proper string for the current locale. It matches the current locale and platform to one of the object's properties and returns the associated string. On a German system, for example, the property `de` provides the string `"Ja"`.

When your script uses localization to provide language-appropriate strings for UI elements, it should also take advantage of the Automatic Layout feature. The layout manager can determine the best size for each UI element based on its localized `text` value, automatically adjusting the layout of your script-defined dialogs to allow for the varying widths of strings for different languages.

Variable values in localized strings

The `localize` function allows you to include variables in the string values. Each variable is replaced with the result of evaluating an additional argument. For example:

```
today = {
  en: "Today is %1/%2.",
  de: "Heute ist der %2.%1."
  };
d = new Date();
Window.alert (localize (today, d.getMonth()+1, d.getDate()));
```

Enabling automatic localization

If you do not need variable replacement, you can use automatic localization. To turn on automatic localization, set the global value:

```
$.localization=true
```

When it is enabled, you can specify a localization object directly as the value of any property that takes a localizable string, without using the `localize` function. For example:

```
btnText = { en: "Yes", de: "Ja", fr: "Oui" };
b1 = w.add ("button", undefined, btnText);
```

The `localize` function always performs its translation, regardless of the setting of the `$.localize` variable. For example:

```
//Only works if the $.localize=true
b1 = w.add ("button", undefined, btnText);
```

```
//Always works, regardless of $.localize value
b1 = w.add ("button", undefined, localize (btnText));
```

If you need to include variables in the localized strings, use the `localize` function.

6 Bridge DOM Object Reference

This chapter provides a complete reference for the objects of the The Bridge Document Object Model (DOM). The Bridge DOM objects are presented alphabetically. For each object, complete syntax details are provided for the constructor, properties, and functions.

App Object	The Bridge application.
Dialog Object	A dialog that displays an HTML page.
Document Object	A Bridge browser window.
Event Object	A user-interaction event.
Favorites Object	An array of the thumbnails shown in the Favorites pane.
Metadata Object	Access to file metadata through a thumbnail.
NavBar Object	A configurable navigation bar that can display user-interface controls.
Preferences Object	Access to application preferences.
PreferencesDialog Object	Access to the Preferences dialog.
Thumbnail Object	A navigable node representing a file, folder, or web page.

Other available objects are discussed separately:

- Chapter 7, "File and Folder Object Reference," describes the ExtendScript `File` and `Folder` objects that provide portable access to the file system.
- Chapter 8, "ScriptUI Object Reference," describes the ExtendScript ScriptUI objects that provide user-interface capability, including the MenuElement Object that allows you to extend Bridge menus.
- Chapter 9, "Interapplication Communication with Scripts," describes the `BridgeTalk` class and message object that provides the ability to communicate among Adobe Creative Suite 2 applications using JavaScript.
- Chapter 10, "ExtendScript Tools and Features," describes various ExtendScript utilities, including:
  - For debugging, the Dollar ($) Object and ExtendScript Reflection Interface
  - The Global localize function and the localization object that allow you to provide language versions of displayed strings for different locales.
  - The UnitValue Object for specifying and working with measurement values

App Object

The `App` object represents the Bridge application. A single global instance is created on startup; access it using the `app` global variable.

There is only one on `App` object. Multiple Bridge browser windows are represented by instances of `Document`, and can be accessed with the `app.document` or `app.documents` properties.

App object properties

`displayDialogs`	String	The policy for the display of modal dialogs. Read/Write. One of: `all` (default): Modal dialogs should always be displayed. `none`: Modal dialog should never be displayed. `error`: Only dialogs that report an error to the user should be displayed.
`document`	Document	The active (top-most) Document Object, representing the active Bridge browser window. Read/Write.
`documents`	Array of Document	A collection of Document Objects representing the set of all open Bridge browser windows. Read/Write.
`eventHandlers`	Array of EventHandler	A collection of event handlers installed by scripts. Add an event handler to this array to register it with Bridge. Registered handler functions are called when any user-interaction event is triggered. See Event Handling in Bridge. Read/Write. Each event handler is specified by a JavaScript object with one property, the handler function name: `{ handler: fnName}` The handler function takes one argument, an Event Object, and returns a result object `{handled: boolean}`. ● When `true`, the event has been completely handled and Bridge does not look for more handlers or call the default handler. ● When `false` (or when the handler returns `undefined`), Bridge continues to call registered handlers, or if there are no more, calls the default handler.
`favorites`	Favorites	The top-level object for the navigation hierarchy displayed in the Favorites pane. This Favorites Object is an array of Thumbnail Objects. Read only.
`language`	String	The display name of the language for the current locale, as configured by the operating system. This is the name as it appears in the Preferences dialog. Read only.

locale	String	The Adobe locale code for the current locale, as configured by the operating system. Read only. An Adobe locale code consists of a 2-letter ISO-639 language code and an optional 2-letter ISO 3166 country code separated by an underscore. Case is significant. For example, `en_US`, `en_UK`, `ja_JP`, `de_DE`, `fr_FR`.
name	String	The CS2 application specifier for this application, "`bridge`". Read only.
preferences	Preferences	The Preferences Object, which provides access to the user preferences shown in the Preferences dialog (invoked from the **Edit > Preferences** command). Read only.
version	String	The version number of the Bridge application. Read only.

App object functions

beep `app.beep ()`	Calls on the operating system to emit a short audio tone. Returns `undefined`.
browseTo `app.browseTo (`*`path`*`)`	Opens a new Bridge browser window and navigates to the specified file or folder. ● For a folder, the Folders pane shows the hierarchy of containment ending at this folder. The Content pane shows the contents of this folder, and no file is selected. ● For a file, the Folders pane shows the hierarchy of containment for the file. The Content pane shows the contents of the containing folder, with this file scrolled into view and selected. ● If `path` specifies a file or folder that does not exist on disk, the browser navigates to a default path; Desktop in Windows, User Home in Mac OS. Creates the new Document Object for the window and makes it the current `app.document`. Returns `undefined`.

`path`

A File Object orFolder Object, or the path to a file or folder in platform-specific or portable path format (URI notation). For example:

```
// using Windows platform-specific notation
app.browseTo("C:\\MyFolder");
// using URI notation
app.browseTo("/c/MyFolder/MyFile");
// the Folder object returns URI notation
app.browseTo(Folder.selectDialog("Browse to?"));
```

For more information on path notations, see Specifying Paths.

Note: There are two ways to browse programmatically from a Bridge script:

- Call the App Object's browseTo function. This creates a new browser window and sets it to display the location you specify.
- Set the Document Object's thumbnail property. This causes the current Folders pane to show and select the specified node.

Starting the Bridge application from a script also opens a browser window. If you use an interapplication message to call the `browseTo` function from a script that also starts the Bridge application, you can inadvertently open two browser windows. To avoid this, use code such as the following:

```
if( BridgeTalk.isRunning('bridge') )
    // create new browser window if needed
    bt.body = "if( app.documents.length == 0 )" +
              " app.browseTo('" + target + "'); " +
              "else" +
              " app.document.thumbnail = new Thumbnail('" + target + "');";
else //reuse window created by upcoming launch
    bt.body = "app.document.thumbnail = new Thumbnail('" + target + "');";
```

For details of how to send and receive interapplication messages, see Chapter 9, "Interapplication Communication with Scripts."

`buildFolderCache` `app.buildFolderCache (path[, recurse])`	Forces Bridge to create thumbnail images for the specified folder, and optionally for all subfolders. These are stored in a cache file in the folder to which they apply. Returns `undefined`.
`path`	The folder. A Folder Object, Thumbnail Object for a folder, or string of the form: `"browseProtocol://pathSpecifier"` If this specifies a file, the cache is built for the containing folder.
`recurse`	Optional. When `true`, builds the thumbnail cache recursively for all subfolders. Default is `false`.
`cancelTask` `app.cancelTask (taskId)`	Cancels a task that has been scheduled using scheduleTask. Returns `undefined`.
`taskId`	The task ID number, as returned from `app.scheduleTask`.
`hide` `app.hide ()`	In Mac OS, performs the platform-specific hide gesture. In Windows, does the equivalent of `app.document.minimize()`. Returns `undefined`.
`preflightFiles` `app.preflightFiles (files)`	For each specified file or folder, if it refers to a resource that does not have a local copy (such as the files referenced by VersionCue nodes), downloads the specified resource. Returns `true` on success.
`files`	An array of strings, each of which is a file or folder specification. See Specifying Paths.
`purgeAllCaches` `app.purgeAllCaches ()`	Purges the thumbnail caches for all folders. See also buildFolderCache and purgeFolderCache. Returns `undefined`.
`purgeFolderCache` `app.purgeFolderCache ([path])`	Purges the thumbnail caches for the specified folder. See also buildFolderCache and purgeAllCaches. Returns `undefined`.

path	Optional. The folder to purge. A Folder Object, Thumbnail Object for a folder, or string of the form: `"browseProtocol://pathSpecifier"` If this specifies a file, the cache is purged for the containing folder. If not supplied, purges all folder caches.
quit `app.quit ()`	Shuts down the Bridge application. All Bridge browser windows are closed. Returns `undefined`.
registerBrowseScheme `app.registerBrowseScheme (name[, browseHandler])`	Registers a new script-defined browse scheme with Bridge. For additional information, see Script-Defined Browse Schemes. Returns `true` on success, `false` if the scheme was previously registered.
name	The name of the browse scheme protocol. Used in the protocol portion of the `path` argument when creating a Thumbnail Object object that uses this browse scheme. For example, if the browse scheme name is "`myScheme`" the `path` argument is "`myScheme://filename`",
browseHandler	Optional. Used internally.
scheduleTask `app.scheduleTask (script, delay[, repeat])`	Executes a script after a specified delay. The script can be executed repeatedly, stopping when it returns `undefined`, or when you cancel the task using cancelTask. Returns the task ID number, which can be used to cancel the scheduled task. For an example, see Scheduling tasks from callbacks.
script	A string containing the script to be run.
delay	A number of milliseconds to wait before executing the script. If 0, waits the default number of milliseconds, which is 10.
repeat	Optional. When `true`, execute the script repeatedly after each elapsed delay. Stop when a script execution returns `undefined`, or when this task is cancelled by calling `app.`cancelTask. Default is `false`, which means execute the script only once.
system `app.system (commandLine)`	Issues the argument to the operating system, as if it were entered on the command line in a shell. Control does not return to Bridge until this function returns. Returns `undefined`.
commandLine	The command to pass to the operating system.

Dialog Object

Represent's a script-defined dialog that displays HTML user-interface controls. This allows some additional flexibility, as an alternative to that available through the ScriptUI Window Object.

Note: The Bridge-DOM dialog object can contain *only* HTML controls; it cannot contain ScriptUI controls. Similarly, the ScriptUI `window` object can contain only ScriptUI controls, not straight HTML.

Dialog object constructor

To create a new Bridge dialog object, use the `new` constructor:

`new Dialog (`*`path`*`);`

path	A string containing the path and file name of the HTML page to display in the dialog.

Dialog object properties

active	Boolean	When `true`, the dialog is visible and frontmost on the screen, when `false` it is not. Set to `true` to make a dialog active. Read/write.
closing	Boolean	When set to `true`, the dialog closes, first invoking the `willClose` and `doClose` callbacks, if supplied. Read/write.
height	Number	The height of the dialog in pixels. Read/write.
modal	Boolean	When `false` (the default), the dialog is modeless, meaning that the invocation function returns immediately, but the dialog stays up until dismissed. A modeless dialog does not prevent input in other windows. Read/write. When `true`, the dialog is modal, meaning that it retains the keyboard focus while it is up, and the user must dismiss it before taking any other action in the application. **Note:** Modal Bridge dialogs are not supported in this release.
title	String	The text displayed in the dialog's title bar. Read/write.
width	Number	The width of the dialog in pixels. Read/write.

Dialog object functions

center *dialogObj*`.center ()`	Centers the dialog on the screen.
close *dialogObj*`.close ()`	Forces the dialog to close, without invoking the `willClose` or `doClose` callbacks. Returns `undefined`. Use this to close a modeless dialog from elsewhere in the Bridge script, in response to some external circumstance. From within the dialog's HTML page, set `closing` to `true` instead.

execJS *dialogObj*.execJS (*script*)	Executes a JavaScript function that is defined within the HTML page displayed in the dialog. If the page that defines the function is not currently displayed, causes a run-time error. **Note:** Do not call this method from a dialog callback function. This attempts to re-enter the JavaScript engine, which is not allowed, and causes Bridge to hang. A callback can, instead, schedule a task using `app.scheduleTask`, and call `execJS` from the function associated with the task. See Scheduling tasks from callbacks. Returns the result of the executed JavaScript function, which must be a Boolean, Number, or String, or `null`.
script	A string containing a script to execute. This typically contains the name and arguments of the JavaScript function to execute, but can have multiple statements, including variable declarations, assignments and so on.
open *dialogObj*.open (*callbacks*)	Opens a modeless dialog, which means that a user can take other actions in the Bridge user interface, as well as manipulating the dialog. Returns `undefined`. When you open a dialog box with the `open` method, you can subsequently close it with the `close` method, or by setting the `closing` property to `true`.

callbacks	A JavaScript object containing the function definitions for one or more callbacks, in the form:
	`{ fnName1: function( [args] ) { fn1_definition },`
	`  fnName2: function( [args] ) { fn2_definition }`
	`  ...`
	`}`
	These functions are available to the code in the HTML page, which can invoke them using the `call` function. They run in Bridge's ExtendScript engine, and can use Bridge DOM objects. See Communicating with Bridge from dialog JavaScript.
	You can provide special callback functions named `willClose` and `onClose`., which are invoked when the user closes the dialog. See Displaying HTML in Bridge Dialogs.

place *dialogObj*.place (*x, y*)	Places the dialog on the screen at the specified position.
x, y	A position expressed as a percentage offset from the origin (left top) of the screen. Real numbers between 0.0 and 1.0. Thus, (0.0, 0.0) specifies the left top, (1.0, 1.0) the right bottom, and (0.5, 0.5) the center of the screen. Negative values and values greater than 1 are allowed, but are normalized to on-screen values. For example, (-0.5, 17.25) is the same as (0.5, 0.25).
print *dialogObj*.print ()	Invokes the platform-specific Print dialog box to print the HTML page displayed in this dialog. Returns `true` on success.

Document Object

Represents a Bridge browser window.The user can create multiple browser windows by selecting the **File > New Window** command. For each Bridge browser window, there is one `Document` instance.

- Access the object for the active Bridge browser window using `app.document`
- Access an array of objects for all open Bridge browser windows in `app.documents`

For a discussion of how the parts of the browser window map to JavaScript objects, see The Bridge DOM and the Bridge Browser Window.

Document object properties

allowDrags	Boolean	When `true` (the default), drag-and-drop of thumbnails is allowed in this browser window. When `false`, thumbnails cannot be dragged within or from this browser window.
contentPaneMode	String	The type of content displayed in the Content pane. One of: `filesystem` (default): The Content pane displays files and folders. `web`: The Content pane displays a web page.
context	Thumbnail	The Thumbnail Object a user has right-clicked to invoke a context menu. Otherwise `undefined`. Read only.
id	Number	Read only. A unique identifier for the browser window, valid for the life of the window. It is possible for more than one `Document` object to reference the same window.

jsFuncs	Object	A JavaScript object containing the function definitions for one or more callbacks, in the form:
		`{ fnName1: function( [args] ) { fn1_definition },`
		`  fnName2: function( [args] ) { fn2_definition }`
		`  ...`
		`}`
		These functions are available to the code in an HTML page displayed in the Content pane, which can invoke them using the `call` function. They run in Bridge's ExtendScript engine, and can use Bridge DOM objects. See Displaying HTML in Bridge. Read/Write.

maximized	Boolean	When `true`, this Bridge browser window is in the zoomed or maximized state. Read only.
minimized	Boolean	When `true`, this Bridge browser window is in the collapsed or minimized state. Read only. **Note:** In Mac OS, a window can be in the zoomed state, and still be minimized. If both `maximized` and `minimized` are `true`, call the document's restore method to un-zoom the window.

navbars	NavBar	Contains the predefined NavBar Objects for the configurable navigation bars.
		● To access the navigation bars that can be shown when the Content pane displays a web page:
		`var myTopBar = docObj.navbars.web.top`
		`var myBtmBar = docObj.navbars.web.bottom`
		● To access the navigation bars that can be shown when the Content pane displays files and folders:
		`var myTopBar = docObj.navbars.filesystem.top`
		`var myBtmBar = docObj.filesystem.top`
		Any of the four bars can be configured to display ScriptUI or HTML UI controls. All are hidden by default.

noItems	String	Text to be displayed in the Content pane when the selected thumbnail is for an empty folder. The default is "No Items to Display". Read/Write.

owner	String	The Adobe Creative Suite 2 application that created or first activated this browser window, if it was not Bridge. A CS2 application specifier, such as:
		`golive`
		`illustrator`
		`indesign`
		`photoshop`
		For details of application specifier format, see Application and Namespace Specifiers.

previewLooping	Boolean	When `true`, thumbnails for video files are displayed in the Preview pane with automatic looping. Read/Write.
selections	Array of Thumbnail	The Thumbnail Objects for the currently selected files in the Content pane of this document. Read only. Change the selections using the Document Object's select, selectAll, deselect and deselectAll methods. A script should wait until the loaded event has occurred before making calls to document selection methods.
showThumbnailName	Boolean	When `true`, thumbnail names are displayed in the Content pane. This overrides the ShowName preference value. Read/Write.

sorts	Array	How the thumbnails in the content pane are sorted. An array containing one JavaScript object:
		`{ type:category, reverse:boolean }`
		The `type` value is one of:
		`"user"`
		`"name"`
		`"date-created"`
		`"date-modified"`
		`"label"`
		`"vc-status"`
		`"rating"`
		`"filesize"`
		`"filetype"`
		`"dimensions"`
		`"resolution"`
		`"colorprofile"`
		The `reverse` value is `true` if the thumbnails are sorted in reverse order in the given category.
		For example, to sort in reverse by creation date:
		`app.document.sorts[0].type = "date-created"`
		`app.document.sorts[0].reverse = true;`

status	String	The text displayed in the document's status line at the bottom of the Content pane. Read/Write.
thumbnail	Thumbnail	The Thumbnail Object for the node currently selected in the Folders pane. Read/Write. Setting this value navigates to and selects the corresponding node in the Folders pane. The selected node is displayed in the Content pane according to its displayMode. **Note:** The `document.thumbnail.children` array is not populated until the loaded event has occurred for the document.

thumbnailViewMode	String	The view mode of the Content pane, as selected by the **View** menu. Read/Write. One of:
		`thumbnails`
		`details`
		`alternates`
		`versions`
		`filmstrip`

visible	Boolean	When `true`, the Browser window is expanded, as opposed to being minimized or collapsed. Setting visible to `false` collapses the window. Read/Write.

`visibleThumbnails`	Array of Thumbnail	Read only. An array of Thumbnail Objects that are currently shown in the Content pane. The array is ordered according to the current sort order, and contains only thumbnails whose `visible` property is `true`.

`visitUrl`	function	A callback function that is called when the Content pane is about to open a URL. Allows the script to approve or redirect the browser. The function takes the URL as an argument, and should return an object with these properties:
		`result`: When `false`, Bridge does not open the new URL. When `true`, it opens the passed URL or a different URL as provided in this object.
		`url`: When present, a URL string that replaces the passed URL.
		`toHistory`: When `false`, the passed or provided URL is not added to the browser's history list. Default is `true`.
		For example, this confirms a link with the user:
		`var myFn = function(url) {`
		`  if( Window.confirm( "Proceed to " +url+ " ?" ) )`
		`    return {result:true};`
		`  else`
		`    return {result:false};`
		`  }`
		`app.document.visitUrl = myFn;`
		This example replaces a link to an unwanted page with an application-specific help page:
		`var helpPageFn= function(url) {`
		`  if( url == "unwanted_page" ) )`
		`    return {result:true, url:"my_help_page",`
		`            toHistory:false};`
		`  else`
		`    return {result:true};`
		`  }`
		`app.document.visitUrl = helpPageFn;`
		Within the context of this function, the implicit `this` variable references this Document Object. For example:
		`var myFilter = function(url) {`
		`  Window.alert(this.thumbnail.displayPath);`
		`  return {result:true, url:url};`
		`}`
		Note: This function is also called when the Content pane switches from a web page view to a filesystem view. In this case, the URL passed to the function is "`about:blank`".

Document object functions

`bringToFront` *`docObj`*`.bringToFront ()`	Makes this browser window the topmost active window in the windowing system. Returns `undefined`.
`close` *`docObj`*`.close ()`	Closes this browser window. Returns `undefined`.

deselect *docObj*.deselect(*thumbnail*)	If the specified thumbnail is a child of this document and is selected, removed it from the selections array and deselects it in the browser window. Returns `true` if the thumbnail was deselected. A script should wait until the loaded event has occurred before making calls to document selection methods.
thumbnail	The Thumbnail Object for the node to deselect.
deselectAll *docObj*.deselectAll ()	Removes all members from the selections array and deselects all thumbnails in the browser window. Returns `undefined`. A script should wait until the loaded event has occurred before making calls to document selection methods.
execJS *docObj*.execJS (*script*)	Executes a JavaScript function that is defined within the HTML page displayed in the Content pane when a thumbnail with `displayMode=web` is selected. If the page that defines the function is not currently displayed, causes a run-time error. **Note:** Do not call this method from a jsFuncs callback function. This attempts to re-enter the JavaScript engine, which is not allowed, and causes Bridge to hang. A callback can, instead, schedule a task using `app.scheduleTask`, and call `execJS` from the function associated with the task. See Scheduling tasks from callbacks. Returns the result of the executed JavaScript function, which must be a Boolean, Number, or String, or `null`. For an example, see Executing script functions defined on HTML UI pages.
script	A string containing a script to execute. This typically contains the name and arguments of the JavaScript function to execute, but can have multiple statements, including variable declarations, assignments and so on.
maximize *docObj*.maximize ()	Maximizes or zooms this browser window. Returns `undefined`.
minimize *docObj*.minimize ()	Minimizes or docs this browser window. Returns `undefined`.
refresh *docObj*.refresh ()	Refreshes the display of this browser window. Returns `undefined`.
resetToDefaultWorkspace *docObj*.resetToDefaultWorkspace ()	Restores the default configuration of the tabbed panes in this browser window. The equivalent of choosing **Window** > **Workspace** > **Reset**. Returns `undefined`.
restore *docObj*.restore ()	Restores this browser window after it has been minimized. In Windows, makes it user-sizeable. In Mac OS, returns it to the user-configured size. Returns `undefined`.

`reveal` *`docObj`*`.reveal(`*`thumbnail`*`)`	Causes the Content pane (not the Folders or Favorites pane) to show the specified thumbnail, scrolling the display if necessary to make it visible. Does not select the Thumbnail. Returns `undefined`.
`thumbnail`	The Thumbnail Object for the node to scroll to.
`select` *`docObj`*`.select(`*`thumbnail`*`)`	If the specified thumbnail is a child of this document and is not selected, adds it to the selections array and selects it in the Content pane. This is the same as selecting the icon in the Content pane with CONTROL-click. Returns `true` if the thumbnail was selected. A script should wait until the loaded event has occurred before making calls to document selection methods.
`thumbnail`	The Thumbnail Object for the node to select.
`selectAll` *`docObj`*`.selectAll ()`	Adds all child Thumbnail Objects of the current thumbnail (`document.thumbnail`) to the selections array, and selects them in the Content pane. This is the same as typing CONTROL-a in the Content pane. Returns `undefined`. A script should wait until the loaded event has occurred before making calls to document selection methods.

Event Object

Represents a user-interaction event, such as clicking a thumbnail. Bridge creates an `event` object whenever one of the triggering events occurs, and passes it to any event handlers that you have registered with the App Object's eventHandlers property. The only way to access an `event` object is as the argument to such an event-handling function. See Event Handling in Bridge for details of how to define and register these functions.

The object with which the user interacted to generate the event is called the *target object* of that event. Different target object types are associated with different types of events, as listed in Event Object Types.

The `event` object defines no functions.

Event object properties

appPath	String	When the `type` is openWith, the platform-specific path to the selected opening application. Otherwise `undefined`. Read only.
document	Document	When the target object is a Thumbnail Object, the Document Object for the browser window in which the event occurred. Otherwise `undefined`. Read only.
favorites	Favorites	When `location` is `favorites`, the Favorites Object for the pane in which the event occurred. Otherwise `undefined`. Read only.
isContext	Boolean	When the target object is a Thumbnail Object, and the type is select, this value is `true` if the event was generated by a right-click (the gesture that normally brings up a context menu). Otherwise `false`.
location	String	The location at which the event occurred. This value helps to distinguish events of the same type than can be triggered in different ways. One of: `app`: The target object is the App Object and the event was generated for an interaction with the operating system. `document`: The target object is a Thumbnail Object and the event was generated for an interaction in the Folders pane, or the target object is a Document Object and the event was generated for an interaction with the windowing environment. `favorites`: The target object is a Thumbnail Object and the event was generated for an interaction in the Favorites pane. `filesystem`: The target object is a Document Object and the event was generated for an interaction with the file system. `prefs`: The target object is the PreferencesDialog Object and the event was generated in the Preferences dialog. `web`: The target object is a Document Object and the event was generated for an interaction with the Internet. In this case, `event.url` contains the URL of the page. Read only.

`object`	Thumbnail, Document, App, PreferencesDialog	The target object of the event; that is, the object that generated the event. Read/Write.
`type`	String	The type of action that triggered the event. Different types of events that are supported for each type of target object; see Event Object Types. Read only.
`url`	String	When `location` is `web`, the URL of the web page. Read only.
`where`	String	When `location` is `favorites`, one of: `standard`: The target object is a predefined member of the `Favorites` array. `user`: The target object is a user-added member of the `Favorites` array. Otherwise `undefined`. Read only.

Event Object Types

Events of different types are triggered for different target objects. All types are described here according to the target object.

App events

When an application event occurs, the `event` object has the following property values:

- The target, *`eventObj`*`.object`, is the App Object
- The location, *`eventObj`*`.location`, is the string `app`.
- The type, *`eventObj`*`.type`, is one of these event types:

`close`	Generated when the Bridge application has received a request to terminate, but has not yet started the process. If the handler returns a `handled` value of `true` in the result object, the termination is cancelled. To query the user, you can set this with the return value of `Window.confirm`. For example: `return { handled:  Window.confirm("Really quit?") };`
`destroy`	Generated when the Bridge application terminates. Occurs when the user exits from Bridge by selecting the **File > Exit** command, when the user closes the final open document, or when a script calls the App Object's quit function. The handler cannot override the default shutdown behavior, but it can take additional actions before the shutdown completes.

Document events

You cannot define event handlers that override the default behavior of Document events. You can, however, write an event handler to take additional actions prior to the event.

When a document event occurs, the `event` object has the following property values:

- The target, *eventObj*`.object`, is a Document Object
- The location, *eventObj*`.location`, can be `app`, `web`, or `document`, depending on the type.
- The type, *eventObj*`.type`, is one of these event types:

`complete`	Location is `web`. Generated when the Content pane successfully displays a web page. The `event` object's url property contains the URL of the page.
`create`	Location is `app`. Generated when a new document is created. Occurs when the user selects the **File > New Window** command, or when a script creates a new document with a constructor call.
`deselect`	Location is `document`. Generated when the OS window focus is removed from the browser window.
`destroy`	Location is `app`. Generated when a document is destroyed. Occurs when the user selects the **File** > **Close Window** command in the UI, when a script closes a document using the Document Object's close method, or when Bridge closes a document if the application is terminated.
`empty`	Location is `web`. Generated when the Content pane tries to display a web page with no content. The `event` object's url property contains the URL of the page.
`failed`	Location is `web`. Generated when the Content pane tries to display a web page and fails. The `event` object's url property contains the URL of the page.
`loaded`	Location is `filesystem`. Generated when the Content pane has finished displaying thumbnail icons for all files. The `Document.thumbnail.children` array is not populated until this event has occurred for the document. A script should wait until this event has occurred before making calls to document selection methods such as select and deselect.
`loading`	Location is `web`. Generated when the Content pane begins trying to display a web page. The `event` object's url property contains the URL of the page.
`open`	Location is `document`. Generated when the user double-clicks a thumbnail or group of thumbnails in the Content pane, after the thumbnail select event. The Document Object's selections property contains the Thumbnail Object or objects. The default action is for the selected item or items to be opened in the Content pane.
`select`	Location is `document`. Generated when the Document window gains the OS window focus.
`stopped`	Location is `web`. Generated when the Content pane is trying to display a web page, and stops before the page is fully loaded. The `event` object's url property contains the URL of the page.
`uploading`	Location is `web`. Generated when the Content pane begins uploading content to a web page via a POST request. The `event` object's url property contains the URL of the target page.

Thumbnail events

When a thumbnail event occurs, the `event` object has the following property values:

- The target, *`eventObj`*`.object`, is a Thumbnail Object.
- The location, *`eventObj`*`.location`, is `document` for an interaction with the Folders or Content pane, or `favorites` for an interaction with the Favorites pane.
  - If `location` is `favorites`, the `favorites` property contains the Favorites Object and the `which` property reflects whether the target thumbnail is a predefined or user-defined member of the favorites array.
- The *`eventObj`*`.document` property contains the Document Object for the browser window in which the event occurred.
- The type, *`eventObj`*`.type`, is one of these event types:

`add`	Location is `favorites`. Generated when the user adds a new node to the Favorites pane.
`deselect`	Location is `document`. Generated when focus is removed from a thumbnail. After a user has clicked a thumbnail `A`, this event occurs with thumbnail `A` as the target when the user clicks on another object, either another thumbnail in the Content pane, or a different folder in the Folders pane. It also occurs when the user chooses the **Edit > Deselect All**, and when a script calls the Document Object's deselect or deselectAll method.
`hover`	Location is `document`. Generated when the cursor hovers over a thumbnail. Your handler can override the text displayed in the tooltip box. Return the text to be displayed in the result object property `tipText`.
`modify`	Location is `favorites`. Generated when the user modifies new node to the Favorites pane by adding a subnode to it.
`move`	Location is `favorites`. Generated when the user changes the position of a node in the Favorites pane.
`open`	Location is `document`. Generated when a thumbnail in the Content pane is opened with an application other than Bridge. Occurs when the user successfully opens a thumbnail with the **File > Open** command, or by double-clicking, or when a script calls the Thumbnail Object's `open` method. **Note:** Opening a folder does not generate this event, because Bridge is the application opening the folder.
`openWith`	Location is `document`. Generated when a user makes a selection of thumbnails in the Content pane, then selects an application from the **Open With** submenu of the **File** or context menu. The object provides a platform-specific path string to the selected application.
`preview`	Location is `document`. Generated when an image thumbnail in the Content pane is selected. The handler can return an object in which the `result` value is an array containing text captions to display under the image in the Preview pane. For example: `{ handled: true, result: ["my image", "new preview caption"] }` The preview caption can be modified this way for images displayed in filmstrip view as well.

remove	Location is `favorites`. Generated when the user removes a node from the Favorites pane.
select	Location is `document`. Generated when a thumbnail in the Content pane is selected. Occurs when a user clicks a specific thumbnail in the Content pane, or chooses the **Edit > Select All** or **Edit > Select Labeled** commands, and when a script calls the Document Object's select or selectAll method. If the selection occurs through a right-button click, the `event` object's isContext value is `true`. If the user double-clicks the thumbnail, this event is followed by a document open event.

PreferencesDialog events

You cannot override the default behavior of a Preferences dialog event. You can, however, write an event handler to take additional actions prior to the default action, such as adding a panel that reflects your own preferences, and interpreting the results from that panel.

When an Preferences dialog event occurs, the `event` object has the following property values:

- The target, *eventObj*`.object`, is the PreferencesDialog Object
- The location, *eventObj*`.location`, is the string `prefs`.
- The type, *eventObj*`.type`, is one of these event types:

cancel	Generated when the user clicks **Cancel** in the Preferences dialog.
create	Generated when the user invokes the Preferences dialog.
destroy	Generated when the user closes the Preferences dialog using the window frame's close button.
ok	Generated when the user clicks **OK** in the Preferences dialog.

Favorites Object

Represents the navigation nodes that appear in the Favorites pane in the Bridge browser. The `Favorites` object is itself an array of Thumbnail Objects.

While the Folders pane shows the full navigation hierarchy, with all folders and subfolders that Bridge can access, the Favorites pane shows only certain top-level folders and one level of subfolders. Subfolders in the Favorites pane can be, but are not necessarily, `children` of the `Thumbnail` for the parent node.

Access the `Favorites` object through the App Object's `favorites` property:

```
currentFavorites = app.favorites
```

Favorites object properties

`length`	Number	The number of Thumbnail Objects in the Favorites pane.
`section`	String	Sets the section of the Favorites pane for the next node operations in the immediate scope. The value does not persist. One of: `standard` (default): The top section of the Favorites pane containing predefined nodes. `user`: The bottom section of the Favorites pane containing user-selected nodes.

Favorites object functions

`addChild` *`favoritesObj`*`.addChild (`*`parentNode, subNode`*`)`	Inserts a new subnode into the current section of the favorites array, and updates the Favorites pane to show the new node below its parent when the parent is selected. Returns `true` on success. If the specified parent node is not in favorites array, returns `false` and does not add the subnode.
`parentNode`	The Thumbnail Object for the parent node. Must be a root node in the favorites array.
`subNode`	The Thumbnail Object for the subnode. This node can be, but does not need not to be a child of the parent `Thumbnail`. It is not added to the parent's `children` array.
`clearAll` *`favoritesObj`*`.clearAll ()`	Deletes all the nodes from the current section of the favorites array and updates the Favorites pane. Returns `undefined`.
`insert` *`favoritesObj`*`.insert (`*`thumbnail`*` [, `*`index`*`])`	Inserts a new node into the current sections of the favorites array, and updates the Favorites pane to show the new node at the root level. Returns `true` on success. If the referenced node is already in the array, returns `false` and does not change the array.
`thumbnail`	The Thumbnail Object for the node to insert.
`index`	Optional. A 0-based index into the existing node array at which to insert the new node, or an object reference for a node in the existing node array. The node is inserted before this existing node. If the value is beyond the end, is not in the existing node array, or is not supplied, the new node is appended to the end of the array.

➤ ***Example***

This example creates four new thumbnails that use a script-defined browse scheme. It inserts two into the favorites array, one at the second position (`app.favorites[1]`), and the other at the end. These appear in the Favorites pane. The remaining thumbnails become children of one of the root nodes. The `Favorites` array length increases by two, since the children are not included in it.

```
app.registerBrowseScheme( "myBrowseScheme");
var top      = new Thumbnail( "myBrowseScheme://root", "My Home" );
var myFiles  = new Thumbnail( "myBrowseScheme://myFiles", "My Files" );
var underTop1 = new Thumbnail( "myBrowseScheme://root/ut1", "UT1" );
var underTop2 = new Thumbnail( "myBrowseScheme://root/ut2", "UT2" );
top.displayPath = "http://www.adobe.com"; //when clicked, show this page
top.displayMode = "web";
myFiles.displayPath = "/C/PersonalFiles"; //when clicked, go to this folder
underTop1.displayPath = "/C/Temp1";
underTop2.displayPath = "/C/Temp2";

favRoot = app.favorites; //the initial array reflects content of Favorites pane
favRoot.section="user"; // modify the user section
favRoot.insert( top, 1 ); // add new thumbnails to Favorites
favRoot.insert( myFiles, top );
top.insert( underTop1 ); // use Thumbnail.insert to add folder children
top.insert( underTop2 );
// add one of the folder children as a subnode in Favorites
favRoot.addChild( favRoot[1], underTop1 );
```

`remove` *`favoritesObj`*`.remove(`*`thumbnail`*`)`	Removes the specified script-defined node from the favorites array and updates the Favorites pane. Returns `true` on success. Scripts cannot access predefined nodes.
`thumbnail`	The Thumbnail Object for the node to remove.

Metadata Object

Allows you to access the Extensible Metadata Platform (XMP) metadata associated with the file node of a Thumbnail Object. This is external data associated with the file, such a copyright owner, author, or camera settings.

Metadata is organized into schemas that group related types of metadata; for example the XMP Rights Management Schema groups metadata associated with ownership and rights, such as copyright and owner. The metadata properties found in a specific schema are accessed via the *namespace* of the schema and the *property name* of the metadata item. For example, the namespace of the XMP Rights Management Schema is `http://www.adobe.com/xap/1.0/rights`, and the copyright property name is `Copyright`.

For more information about XMP metadata, see the XMP specification:

http://partners.adobe.com/asn/tech/xmp/pdf/xmpspecification.pdf

Access the `Metadata` object for a file-type thumbnail through the Thumbnail Object's metadata property:

```
var t = new Thumbnail (File ("/C/mydir/myfile"));
var mdata = t.metadata
```

Metadata object properties

Label	String	Provides programmatic access to an image thumbnail label value. Thumbnail labels are set interactively through the **Label** menu in the menu bar and in the right-click context menu of a thumbnail. The choices that appear in the menu are controlled by a preference, which is in turn accessible through the Preferences Object's Label1 property. For example, if your preferences associate the red flag with the string `Urgent`, the string `Urgent` appears in the **Label** menu (in place of the default string `Red`). When you choose that label for a thumbnail, the string `Urgent` appears in this property and the thumbnail is displayed with a red highlight frame. The string is only displayed with the thumbnail if you choose to display the `Label` metadata value as one of the metadata display lines. You can set a label programmatically by setting this property to any string. Setting a `thumbnail.metadata.Label` value directly allows you to use a label other than the ones defined in the preferences. If the string is not one of the label preferences, it is associated with a white highlight frame. An empty string value means the thumbnail is not labeled.

namespace	String	The current XMP namespace, used to search for XMP properties. Default is the root namespace. Read/Write. To access values in a specific schema, the namespace for that schema must be set before referencing the properties in the schema.
xmpPropertyName	String	Get or set a simple XMP property value for a thumbnail by specifying it as a property of that thumbnail's `metadata` object. Properties are accessed in the current namespace. Read/Write. New simple metadata properties are created and added to the current namespace when a script references a new property name. You can add properties only to currently defined namespaces, not to the root namespace. Property names are case sensitive.

Examples

This script gets the metadata associated with the first thumbnail in the content pane. Once the proper namespace is set, it reads the "Exposure" metadata property associated with that thumbnail.

```
var m = app.document.thumbnail.metadata;
m.namespace = "http://ns.adobe.com/camera-raw-settings/1.0/";
Window.alert( "Metadata.Exposure: " + m.Exposure );
```

This script gets the metadata associated with a specific file. It make the photoshop (IPTC) namespace current, and sets the value for the "Author" property in that namespace. It then creates a new metadata property, "SpecialNotes", which is added to the namespace.

```
tn = new Thumbnail( File("/C/MyFiles/txtFile.txt") );
md = tn.metadata;
md.namespace = "http://ns.adobe.com/photoshop/1.0/";
md.Author = "Jane Smith";
Window.alert("file author: ", md.Author);
md.SpecialNotes = "Special notes for this file.";
Window.alert("Special Notes: ", md.SpecialNotes);
```

This script sets a user-supplied label flag for all currently selected thumbnails.

```
var sels = app.document.selections;
var label = Window.prompt("Label:");
for (i = 0; i < sels.length; i++){
  var t = sels[i];
  var m = t.metadata;
  m.namespace = "http://ns.adobe.com/xap/1.0/";
  m.Label = label;
}
```

This script gets one of the values of a multivalued property. In this case, the `Media` property's value is a JavaScript object with a property `bmsp`:

```
var t = app.document.selections[0];
var m = t.metadata;
m.namespace = "http://ns.adobe.com/StockPhoto/1.0/";
var media = m.Media
var langID = media["Media/bmsp:LanguageID"];
```

Metadata object functions

applyMetadataTemplate *metadataObj*.applyMetadataTemplate (*templateName*, *modType*)	Adds metadata properties to this object that were saved to an XMP template from the FileInfo dialog. Returns `undefined`.
templateName	String, The name of the XMP template. Templates are stored for each user in: • (Windows) `%APPDATA%/Adobe/XMP/Metadata Templates/` • (Mac OS) `/Users/`*`username`*`/Library/Application Support/Adobe/XMP/Metadata Templates/`
modType	The modification type, one of: `append`: Adds to the metadata any property that is in the template but not in the source. If a property in the template already exists in the source, its value is not changed, unless it is an array. For an array, adds members that are in the template but not in the source. If an array member already exists in the source, the value is not changed. `replace`: Adds to the metadata all properties and values that are in the template. If a property in the template already exists in the source, its value is changed to the template value.

NavBar Object

Represents a configurable navigation bar, one of which can be displayed at the top of the Bridge browser window (below the application navigation bar), and one at the bottom (above the status bar). You do not create new `NavBar` objects. Instead, you access the existing objects through the Document Object' s properties:

```
topbarW = app.document.navbars.web.top
btmbarW = app.document.navbars.web.bottom
topbarF = app.document.navbars.filesystem.top
btmbarF = app.document.navbars.filesystem.bottom
```

The bars in `navbars.web` can be shown when the Content pane displays a web page. The bars in `navbars.filesystem` can be shown when the Content pane displays files and folders.

The navigation bars are hidden by default. You can show and hide them by setting the `NavBar` object's `visible` property.

Your script can configure a navigation bar to contain user-interface controls such as push buttons, radio buttons, edit fields, list boxes, and so on. The `NavBar` objects are initially empty. You can either add ScriptUI controls, or reference HTML code that provides the interface controls. You cannot mix the two types of controls; the bar displays either ScriptUI or HTML. For further discussion, see Navigation Bars.

NavBar object properties

file	String	When `type=html`, the URL for the HTML page to display. Read/Write.
height	Number	The height of the navigation bar. (default is 40 pixels). Read/Write.

jsFuncs	Object	A JavaScript object that defines a set of callback functions that access the Bridge DOM, but can be called from within an HTML page displayed in this navigation bar. Used only when `type=html`. Read/Write.
		Each property in the object is a callback function name, and the value is the function declaration:
		`{`
		`fnName1: function( [args] ) { fn1_definition },`
		`fnName2: function( [args] ) { fn2_definition }`
		`}`
		The HTML page displayed by this bar can access the Bridge DOM by invoking one of these callbacks, using the JavaScript function `call`. For example, suppose `jsFuncs` has the value:
		`{ myFn: function(x) { return x > app.document.topNavBar.height } },`
		A script on the displayed HTML page can invoke this function as follows:
		`var toobig = call("myFn", 55);`
		See Displaying HTML in a Navigation Bar.

type	String	The type of user-interface controls displayed in the navigation bar. Read/Write. One of: `scriptui`: Display the ScriptUI controls added with this object's add method. `html`: Display the HTML page specified by `file`.
visible	Boolean	Controls whether the NavBar is visible or not. If true, the navigation bar is visible. Default is false. Read/Write.

NavBar object functions

add

```
navBarObj.add (type
[, bounds, text,
 { creation_props> } ]);
```

Creates and returns a new ScriptUI control or container object and adds it to the children of this navigation bar. Returns `null` if unable to create the object.

For an example, see Displaying ScriptUI elements in a navigation bar.

type	The control type. See Control types and creation parameters.
bounds	Optional. A bounds specification that describes the size and position of the new control or container, relative to its parent. See Bounds object for specification formats. If supplied, this value creates a new Bounds object which is assigned to the new object's `bounds` property.
text	Optional. A string containing the initial text to be displayed in the control as the title, label, or contents, depending on the control type. If supplied, this value is assigned to the new object's `text` property.
creation_props	Optional. The properties of this JavaScript object specify creation parameters, which are specific to each object type. See Control types and creation parameters.

execJS *navBarObj*.execJS (*script*)	Executes a JavaScript function that is defined within the HTML page displayed in the navigation bar when `type=html`. If the page that defines the function is not currently displayed, causes a run-time error. **Note:** Do not call this method from a `NavBar` callback function defined in `jsFuncs`. This attempts to re-enter the JavaScript engine, which is not allowed, and causes Bridge to hang. A callback can, instead, schedule a task using `app.scheduleTask()`, and call `execJS` from the function associated with the task. See Scheduling tasks from callbacks. Returns the result of the executed JavaScript function, which must be a Boolean, Number, or String, or `null`. For an example, see Displaying HTML in a Navigation Bar.
script	A string containing a script to execute. This typically contains the name and arguments of the JavaScript function to execute, but can have multiple statements, including variable declarations, assignments and so on.
print *navBarObj*.print ()	Prints the HTML page displayed in the navigation bar when `type=html`. Does nothing if the HTML is not yet loaded when the call is made, or if `type=scriptui`. Returns `true` on success.

Preferences Object

Allows access to the Bridge application preferences, as viewed in and controlled by Preferences dialog (invoked by the **Edit > Preferences** command).

- Some existing preferences can be set or read by setting or retrieving the associated property value. Not all existing preferences are available in the scripting environment. Those that are available are listed below. Preference values do not take effect until the Bridge application is restarted.
- You can set certain preference values for the current session only. That is, the changes take effect immediately, but do not persist across sessions. The next time the Bridge application is restarted, the global preference value is used.
- A script can create a new preference by simply referencing a new property name in this object. New preferences must be of the type String, Number, or Boolean. Composite types (such as Rect and Point) are retrieved as String objects.

Access the `Preference` object i through the App Object's preferences property:

```
var prefs = app.preferences;
```

Preferences object properties

The following current-view properties allow you to set these styles for a specific Content pane view. They do not change the related global preference, and the changes do not persist beyond the current view:

`extraMetadata`	Array of Number	An array of three values, where each value identifies a metadata property to be displayed beneath a thumbnail icon. Read/Write.
		
		Setting this property is the same as setting the preferences associated with the **Additional Thumbnail Metatdata** drop-down lists and checkboxes in the Preferences/General pane, except that the setting does not persist beyond the current view.
		
		The first value in the array sets the first line of additional metadata, the second value sets the second line, and the third value sets the third line. A number in the corresponding array location identifies the metadata property to be displayed in that line. The numeric values map to metadata properties as follows:
		
		`1: DateCreated`
		`2: DateModified`
		`3: Dimensions`
		`4: Label`
		`5: Author`
		`6: Keywords`
		`7: Copyright`
		`8: ColorMode`
		`9: BitDepth`
		`10: DocumentCreator`
		`11: OpeningApplication`
		`12: Exposure`
		An array value of `undefined` turns off the display of metadata for that line.

`showName`	Boolean	When `true`, the names of thumbnails are displayed beneath the icon in this view. When `false`, they are not. Read/Write. (This is overridden by the document's showThumbnailName value.)

The following properties allow access to existing application preferences. Preference values do not take effect until the Bridge application is restarted:

`BackgroundColor`	Number	In the Preferences/General pane, the preference associated with the **Background** slide bar. Read/Write. The background color is set in the range of 0 - 255, where 0 is black, and 255 is white. Default 186.
`FileSize`	Number	In the Preferences/Advanced pane, the preference associated with **Do not process files larger than: *nnn* MB**. Default 200. Read/Write.
`HideEmptyFields`	Boolean	In the Preferences/Metadata pane, the preference associated with the **Hide Empty Fields** checkbox, `true` when checked. Default `true`. Read/Write.
`Label1` **`Label2`** **`Label3`** **`Label4`** **`Label5`**	String	In the Preferences/Labels pane, the preferences associated with the label colors and their keyboard shortcuts. These preferences control the choices that appear in the **Label** menu in the menu bar and in the right-click context menu for image thumbnails. Read/Write. The preference value is any string. For example, if you associate the red flag with the string `Urgent`, the string `Urgent` appears in **Label** menu (in place of the default string `Red`), in the tooltip for the labeled thumbnail, and in a labeled thumbnail's metadata Label value. The thumbnail is displayed with a red highlight frame. The labeling feature is only available for those thumbnails associated with image files. By default, no labels are set. Labels can be set interactively by choosing from the **Label** menu or programmatically by setting the `Thumbnail.metadata.Label` value to any string. If that string is not one of the preferences, it is associated with a white highlight frame.
`Language`	String	In the Preferences/Advanced pane, the preference associated with **Language**. Read/Write.
`MRUCount`	Number	In the Preferences/Advanced pane, the preference associated with **Number of Recently Visited Folders to Display in the LookIn Popup**. Read/Write.
`ShowLabels`	Boolean	In the Preferences/General pane, the preference associated with **Show Labels**, `true` when checked. Default `true`. Read/Write.
`ShowName`	Boolean	When `true`, the names of thumbnails are displayed beneath the icon. When `false`, they are not. Read/Write.

UseLocalCaches	Boolean	In the Preferences/General pane, the preference associated with **Cache** choices, `true` when **Use a Centralized Cache File** is selected. Default `true`. Read/Write.

anyPropertyName	Number, String, or Boolean	A script-defined preference. Read/Write.
		This example creates a new preference named `mypref` by assigning a value to the property `sample`, then accesses the value by reading the property.
		`app.preferences.mypref = "sample value";`
		`Window.alert("New preference mypref = " +`
		`    app.preferences.mypref);`
		To add your script-defined preference to the Preferences dialog, use the PreferencesDialog Object's addPanel function.
		Note: The script must implement default values and initialization of any private setting stored in the Bridge preferences.

Preferences object functions

clear *prefObj*.clear ([*name*[, *name2*...]])	Removes script-created keys and values from the Bridge preferences, or resets preferences. Returns `undefined`. ● If one or more preference names is passed, each is removed. If you try to access the property for a preference that has been removed, the property returns `undefined`. ● If no preference names are passed, removes all script-defined preferences, and resets all Bridge application preferences to their default values.
name	Optional. One or more names of preferences to remove.

PreferencesDialog Object

Provides access to the Bridge Preferences dialog, allowing you to add a panel to the dialog with your own ScriptUI controls that access and set any application preferences that you have defined by adding properties to the Preferences Object.

You can only access this object as the target of an event. The object is returned in the object property of an Event Object that results from an event in a Preferences dialog. See PreferencesDialog events.

The Preferences dialog is modal, which means that no other Bridge events can occur until the user dismisses it with the **OK** or **Cancel** button, or closes it with the window-frame icon.

- For the **OK** button, the dialog generates an `ok` event. Your handler can collect the values from the controls in your panel, and modify the property values in the `Preferences` object accordingly.
- For the **Cancel** button, the dialog generates a `cancel` event, and for the window-close gesture, it generates a `destroy` event. Your handler can, for example, clean up structures you created for the window.

The object has no properties.

PreferencesDialog object functions

addPanel
prefObj.add (*name*)

Creates and returns a ScriptUI Window Object to be used as a new pane in the Preferences dialog. You can add ScriptUI controls to the window to allow users to access and set preferences that you provide.

name	The name of the new panel, used as the title of the new `window` object.

➤ ***Example***

This example adds a pane to the Preferences dialog that contains a single checkbox, which controls the boolean preference named `myPref`.

```
function doPrefs(dialog) {
  var panel = dialog.addPanel("My Preferences");
  var aBox = panel.add( 'checkbox', [50, 50, 200, 100],  "My Pref",
      { alignment:['center','top'] } );
  aBox.onClick = function() { app.preferences.myPref = aBox.value; };
}

var myHandler = function(event) {
  if (event.type == "create" && event.location == "prefs") {
      doPrefs(event.object);
  }
  return { handled: false };
};

app.eventHandlers.push( { handler: myHandler } );
```

close
prefObj.close (*isOK*)

Closes the Preferences dialog.

isOK	Pass `true` to simulate the user clicking **OK** to close the dialog, `false` for **Cancel**.

Thumbnail Object

Represents a reference to a node in the browser navigation hierarchy. Thumbnail objects can represent:

- Files and folders in the local file system.
- Version Cue nodes
- URLs
- Script-defined navigation nodes associated with script-defined browse schemes.

A thumbnail's *browse scheme* determines what is shown in the Content pane when the user selects that thumbnail in the Folders or Favorites pane. The Content pane can show a filesystem hierarchy or a local or remote web page. A thumbnail can use a predefined or script-defined browse scheme.

Thumbnail object constructor

To create a new `Thumbnail` object, use the `new` constructor:

`new Thumbnail (`*`node`*`[, `*`name`*`]);`

node	The node specifier. One of the following: • A File Object or Folder Object for file or folder that exists on the local file system. If the referenced file or folder does not exist, causes a run-time error. This object becomes the value of the new object's `spec` property. • A `Thumbnail` object. This creates a new `Thumbnail` object that references the same node. See Multiple references to the same node. • A path to a local or remote file, folder, or page, which becomes the value of the new object's `path` property. The path can include a browse-scheme specifier; see Node specifiers below.
name	Optional. A localizable string to use as the display name for the thumbnail icon in the browser window. For script-defined browse schemes, the browse scheme must be registered before the thumbnail is created for the name to take effect. If not supplied, the display name defaults to the `path` or `spec` value. **Caution:** For a `Thumbnail` object associated with aFile Object or Folder Object, using the `name` argument renames the folder or file on disk.

Node specifiers

A node specifier that specifies a predefined or script-defined browse scheme takes the form:

`[`*`browseProtocol`*`://]`*`path`*

The *`browseProtocol`* can be script-defined or predefined. The predefined browse scheme protocols include:

- `http://`, `https://` (for web browsing): Creates a thumbnail with `displayMode="web"` that references the URL in *`path`*, which becomes the value of `displayPath`. When selected, navigates to the web page associated with the URL and displays it in the Content pane. If the URL does not exist, causes a nonfatal error, such as "Could not locate remote server."

A script-defined *browseProtocol* is the name of a browse scheme that you have registered using the App Object's registerBrowseScheme method. In this case, the *path* for the top-level node in the browse scheme hierarchy must be `root`. See Script-Defined Browse Schemes.

➤ **Examples of thumbnail creation**

```
// references a folder
var myLocation = new Thumbnail(Folder("/C/myFolder"));
// a second reference to the same node
var newLocation = new Thumbnail( myLocation );
// references a file, and renames the file on disk
var myFile = new Thumbnail(File("/C/myFolder/file.txt"), "myfile.txt" );
// references a URL
var myURL = new Thumbnail ("http://www.adobe.com");
```

Multiple references to the same node

Multiple `Thumbnail` objects can refer to the same node. In JavaScript terminology, two such objects are equal, but not identical. That is, if you declare two `Thumbnail` objects that point to the same file, the JavaScript equality operator "==" returns `true`, but the identity operator "===" returns `false`. Any arbitrary properties assigned to one of the objects are not be reflected in the other.

This example creates two `Thumbnail` objects that reference the same node, and shows that an arbitrary property defined on one cannot be referenced on the other.

```
var t1 = new Thumbnail(File("/C/Temp/afile.txt");
var t2 = new Thumbnail(File("/C/Temp/afile.txt");

t1 == t2;  // returns true
t1 === t2; // returns false

t1.newNote = "a note for the thumbnail";
alert(t2.newNote);  // t2.newNote is undefined.
```

For a `thumbnail` that references a `File` object, however, you can assign arbitrary data to the `metadata` object, which can be referenced from either object.

```
var t1 = new Thumbnail( File("/C/myFolder/myfile.txt"));
var t2 = new Thumbnail( File("/C/myFolder/myfile.txt"));
t1.newProperty = "arbitrary value";
var val = t2.newProperty; // result is undefined.
//properties created directly in thumbnail are not shared
var md = t1.metadata;
md.namespace = "http://ns.adobe.com/photoshop/1.0/";
md.SpecialNotes = "Special notes for this file.";
// You can access SpecialNotes from either Thumbnail object
t2.metadata.namespace = "http://ns.adobe.com/photoshop/1.0/";
alert("Special Notes: ", t2.metadata.SpecialNotes);
```

The `spec` values of the two thumbnail objects reference different `File` objects, and so are not equal. However, the two `File` objects reference the same file, as shown by inspecting the string value:

```
t1.spec == t2.spec; //returns false
t1.spec.toString() == t2.spec.toString(); // returns true
```

Thumbnail object properties

aliasType	String	If the value of type is alias, the kind of target this thumbnail represents, one of: file folder Otherwise undefined.
children	Array of Thumbnail	An array of Thumbnail objects for the children of this container node. When this object references a folder, the children are the thumbnails that reference the contents of the folder. By default, when the thumbnail is selected in a navigation pane, its children are shown in the Content pane. Read only. **Note:** This array is not populated until the loaded event has occurred for the document. The list of children is cached on the first reference so that subsequent references do not result in further disk access. To ensure that the list is up to date (for example after you have performed operations that may have resulted in children being deleted, added, or renamed) call the refresh method to make sure the list is updated on the next access. You do not need to refresh if you changed the content or properties of a child thumbnail.
container	Boolean	When true, the node is a container. Folder thumbnails, web browser thumbnails, Version Cue thumbnails and thumbnails with script-defined browse schemes are all containers. Only container nodes can appear in the Folders and Favorites panes. Does not indicate whether the node currently has any children, just whether it can have them. Read only.
creationDate	Date	Date the referenced file or folder was created, if it can be determined. Read only.
displayMode	String	The display mode of the Content pane when this thumbnail is selected in a navigation pane. Read/Write. One of: filesystem (default): The Content pane shows the contents of a folder. web: The Content pane shows the HTML page referenced by displayPath.
displayPath	String	The path to a file that is displayed in the Content pane when this thumbnail is selected in a navigation pane. Read/Write. Default is the value of path.
hidden	Boolean	When true, this thumbnail is hidden. When false (the default), it is shown. Read Only
lastModifiedDate	Date	Date the referenced file or folder was last modified, if it can be determined. Read only.

location	String	Whether the thumbnail is associated with a local file-system object or a Version Cue node (which can have both a local and remote replica). One of:
		`local`
		`unknown`
		`VersionCue`

metadata	Metadata	Immediately returns the Metadata Object associated with this thumbnail, if it can be found. Read only.
mimeType	String	The referenced file's MIME type, if it can be determined; otherwise, the empty string. Read only.
name	String	The label displayed for the thumbnail. Read/Write. Default is the `path` value.
parent	Thumbnail	The `Thumbnail` object for the parent node of this thumbnail. The value is `undefined` for thumbnails added to the root level of `app.favorites`. This object is in the `children` array of its parent. Read/Write.
path	String	A node specifier in the form: `[`*browseProtocol*`://]`*path* Set when the object is created, using the first argument to the Thumbnail object constructor. Read only.
spec	File, Folder	A File Object or Folder Object for this thumbnail's node. If the thumbnail does not encapsulate a file or folder, the value is `undefined`. Read only.
synchronousMetadata	Metadata	Waits up to three seconds to return the Metadata Object associated with this thumbnail, if it can be found. Read only.

type	String	The type of node this thumbnail references. One of:
		`file`
		`folder`
		`alias`
		`package`
		`other`

Thumbnail object functions

copyTo

thumbnailObj`.copyTo (`*target*`)`

Creates a new Thumbnail Object that references the same node as this one, and adds it to the target thumbnail's `children` list. Returns `true` on success.

```
var thumbnail =
    new Thumbnail(File.openDialog("Source?"));
var target =
    new Thumbnail(Folder.selectDialog("Target?"));
if (thumbnail.copyTo(target)) {
    Window.alert("copy succeeded");
}
else Window.alert("copy failed");
```

target	A Thumbnail Object to be the parent of the new copy.

moveTo *thumbnailObj*.moveTo (*target*)	Removes this thumbnail from its current parent, and adds it to the target thumbnail's `children` list. Returns `true` on success. **Note:** If the thumbnail refers to an existing file or folder, this moves the referenced file or folder on disk. `var thumbnail = new Thumbnail(File.openDialog("Source?")); var target = new Thumbnail(Folder.selectDialog("Target?")); if (thumbnail.moveTo(target)) { Window.alert("move succeeded"); } else Window.alert("move failed");`
target	A Thumbnail Object to be the parent of the new copy.
open *thumbnailObj*.open ()	Launches the file referenced by this thumbnail in the appropriate application (such as Photoshop for JPEG files). This is the same as choosing **Open** from the **File** or context menu, or double-clicking the thumbnail icon in the Content pane. If this thumbnail references a JSX file, runs the script in its target application, or, if no target is specified, in the ExtendScript Toolkit. See Preprocessor directives. If this thumbnail references a folder, navigates to that folder in the Folders pane—that is, sets `document.thumbnail` to this thumbnail. Returns `true` on success.
openWith *thumbnailObj*.openWith (*appPath*)	Launches the file referenced by this thumbnail in the specified application. Returns `true` on success.
appPath	A platform-specific path string to the application, as returned in `appPath` property of the openWith event object when a user makes a selection of thumbnails in the Content pane, then selects an application from the **Open With** submenu of the **File** or context menu.
refresh *thumbnailObj*.refresh ()	Refreshes the node's internal information to reflect the current state of its referenced file or folder. For non-container thumbnails, returns `true` if the node has changed since the last access. For container thumbnails, marks the `Thumbnail` object so that the next access to the `children` property causes a disk access to update the cached list of children, and returns `true` if the node has been renamed since the last access.

remove *thumbnailObj*.remove ()	Deletes this Thumbnail object, and also deletes the file or folder associated with the thumbnail from the disk. Returns true on success.

➤ **Example**

```
var tn = new Thumbnail(File.openDialog("Delete?"));
if (!tn.remove()){
   Window.alert("Thumbnail deletion failed");
}
```

resolve *thumbnailObj*.resolve ()	If the value of type is alias, returns a Thumbnail object for the target of the alias, or, if the alias cannot be resolved, returns undefined. If the type is not alias, returns this Thumbnail object.

7 File and Folder Object Reference

Overview

Because path name syntax is very different in Windows, Mac OS, and UNIX, the `File` and `Folder` objects are defined to provide platform-independent access to the underlying file system. A `File` object is associated with a disk file; a `Folder` object with a directory or folder.

- The `Folder` object supports file-system functionality such as traversing the hierarchy, creating, renaming, or removing files, or resolving file aliases.
- The `File` object supports I/O functions to read or write files.

`File` and `Folder` objects can be used anywhere a path name is required, such as in properties and arguments for files and folders.

For a description of the pathname syntax and object usage, see Chapter 4, "Using File and Folder Objects." This chapter provides detail about the classes and objects, their properties and methods, and the supported encoding names:

- File Object
- Folder Object
- File and Folder Error Messages
- File and Folder Supported Encoding Names

File Object

Represents a file in the local file system in an platform-independent manner. All properties and methods resolve file system aliases automatically and act on the original file unless otherwise noted.

File object constructors

To create a `File` object, use the `File` function or the `new` operator. The constructor accepts full or partial path names, and returns the new object. The CRLF sequence for the file is preset to the system default, and the encoding is preset to the default system encoding.

```
File ([path]); //can return a Folder object
new File ([path]); //always returns a File object
```

path	Optional. The absolute or relative path to the file associated with this object, specified in platform-specific or URI format; see Specifying Paths. The value stored in the object is the absolute path. The path need not refer to an existing file. If not supplied, a temporary name is generated. If the path refers to an existing folder: • The `File` function returns a `Folder` object instead of a `File` object. • The `new` operator returns a `File` object for a nonexisting file with the same name.

File class properties

This property is available as a static property of the `File` class. It is not necessary to create an instance to access it.

fs	String	The name of the file system. Read only. One of `Windows`, `Macintosh`, or `Unix`.

File class functions

These functions are available as static methods of the `File` class. It is not necessary to create an instance to call them.

decode `File.decode (`*what*`)`	Decodes the specified string as required by RFC 2396 and returns the decoded string.
what	String. The encoded string to decode. All special characters must be encoded in UTF-8 and stored as escaped characters starting with the percent sign followed by two hexadecimal digits. For example, the string "`my%20file`" is decoded as "`my file`". Special characters are those with a numeric value greater than 127, except the following: `/ - _ . ! ~ * ' ( )`
encode `File.encode (`*what*`)`	Encodes the specified string as required by RFC 2396 and returns the encoded string. All special characters are encoded in UTF-8 and stored as escaped characters starting with the percent sign followed by two hexadecimal digits. For example, the string "`my file`" is encoded as "`my%20file`". Special characters are those with a numeric value greater than 127, except the following: `/ - _ . ! ~ * ' ( )`
what	String. The string to encode.
isEncodingAvailable `File.isEncodingAvailable (`*name*`)`	Returns `true` if your system supports the specified encoding, `false` otherwise.
name	String. The encoding name.

openDialog `File.openDialog ([prompt][,select])`	Opens the built-in platform-specific file-browsing dialog in which a user can select an existing file to open. If the user clicks **OK**, returns a `File` object for the selected file. If the user cancels, returns `null`.
prompt	Optional. A string containing the prompt text, if the dialog allows a prompt.
select	Optional. A file or files to be preselected when the dialog opens: ● In Windows, a string containing a comma-separated list of file types with descriptive text, to be displayed in the bottom of the dialog as a drop-down list from which the user can select which types of files to display. Each element starts with the descriptive text, followed by a colon and the file search masks for this text, separated by semicolons. For example, to display two choices, one labeled **Text Files** that allows selection of text files with extensions `.TXT` and `.DOC`, and the other labeled **All files** that allows selection of all files: `Text Files:*.TXT;*.DOC,All files:*` ● In Mac OS, a string containing the name of a function defined in the current JavaScript scope that takes a `File` object argument. The function is called for each file about to be displayed in the dialog, and the file is displayed only when the function returns `true`.
saveDialog `File.saveDialog ([prompt][,select])`	Opens the built-in platform-specific file-browsing dialog in which a user can select an existing file location to which to save this file. If the user clicks **OK**, returns a `File` object for the selected file, and overwrites the existing file. If the user cancels, returns `null`.
prompt	Optional. A string containing the prompt text, if the dialog allows a prompt.
select	Optional. A file or files to be preselected when the dialog opens: ● In Windows, a string containing a comma-separated list of file types with descriptive text, to be displayed in the bottom of the dialog as a drop-down list from which the user can select which types of files to display. Each element starts with the descriptive text, followed by a colon and the file search masks for this text, separated by semicolons. For example, to display two choices, one labeled **Text Files** that allows selection of text files with extensions `.TXT` and `.DOC`, and the other labeled **All files** that allows selection of all files: `Text Files:*.TXT;*.DOC,All files:*` ● In Mac OS, a string containing the name of a function defined in the current JavaScript scope that takes a `File` object argument. The function is called for each file about to be displayed in the dialog, and the file is displayed only when the function returns `true`.

File object properties

These properties are available for `File` objects.

absoluteURI	String	The full path name for the referenced file in URI notation. Read only.
alias	Boolean	When `true`, the object refers to a file system alias or shortcut. Read only.
created	Date	The creation date of the referenced file, or `null` if the object does not refer to a file on disk. Read only.
creator	String	The Mac OS file creator as a four-character string. In Windows or UNIX, value is `"????"`. Read only.
encoding	String	Gets or sets the encoding for subsequent read/write operations. One of the encoding constants listed in File and Folder Supported Encoding Names. If the value is not recognized, uses the system default encoding. A special encoder, `BINARY`, is used to read binary files. It stores each byte of the file as one Unicode character regardless of any encoding. When writing, the lower byte of each Unicode character is treated as a single byte to write.
eof	Boolean	When `true`, a read attempt caused the current position to be beyond the end of the file, or the file is not open. Read only.
error	String	A message describing the last file system error; see File and Folder Error Messages. Setting this value clears any error message and resets the error bit for opened files.
exists	Boolean	When `true`, the path name of this object refers to an existing file. Read only.
fsName	String	The platform-specific name of the referenced file as a full path name. Read only.
hidden	Boolean	When `true`, the file is not shown in the platform-specific file browser. Read/write. If the object references a file-system alias or shortcut, the flag is altered on the alias, not on the original file.
length	Number	The size of the file in bytes. Can be set only for a file that is not open, in which case it truncates or pads the file with 0-bytes to the new length.
lineFeed	String	How line feed characters are written. One of: `windows`: Windows style `mac`: Mac OS style `unix`: UNIX style
modified	Date	The date of the referenced file's last modification, or `null` if the object does not refer to a file on disk. Read only.
name	String	The name of the referenced file without the path specification. Read only.
parent	Folder	The `Folder` object for the folder that contains this file. Read only.

path	String	The path portion of the absolute URI, or the empty string If the name does not have a path. Read only.
readonly	Boolean	When `true`, prevents the file from being altered or deleted. If the referenced file is a file-system alias or shortcut, the flag is altered on the alias, not on the original file.
relativeURI	String	The path name for the referenced file in URI notation, relative to the current folder. Read only.
type	String	The Mac OS file type as a four-character string. In Windows and UNIX, the value is `"????"`. Read only.

File object functions

These functions are available for `File` objects.

close *fileObj*.close ()	Closes this open file. Returns `true` on success, `false` if there are I/O errors.
copy *fileObj*.copy (*target*)	Copies this object's referenced file to the specified target location. Resolves any aliases to find the source file. If a file exists at the target location, it is overwritten. Returns `true` if the copy was successful, `false` otherwise.
target	A string with the URI path to the target location, or a `File` object that references the target location.
createAlias *fileObj*.createAlias (*toFile*, [*isFinderAlias*])	Makes this file into a file-system alias or shortcut to the specified file. The referenced file for this object must exist on disk. Returns `true` if the operation was successful, `false` otherwise.
toFile	The `File` object for the target of the new alias.
isFinderAlias	Optional, Mac OS only. When `true`, the alias is created as a legacy Finder alias. When `false` (the default), the alias is created as a UNIX symlink.
execute *fileObj*.execute ()	Opens this file using the appropriate application (as if it had been double-clicked in a file browser). You can use this method to run scripts, launch applications, and so on. Returns `true` immediately if the application launch was successful.
getRelativeURI *fileObj*.getRelativeURI ([*basePath*])	Returns a string containing the URI for this file or folder relative to the specified base path, in URI notation. If no base path is supplied, returns the URI relative to the path of the current folder.
basePath	Optional. A string containing the base path for the relative URI. Default is the current folder.

open *fileObj*.open (*mode*[, *type*] [, *creator*])	Open the file for subsequent read/write operations. The method resolves any aliases to find the file. Returns `true` if the file has been opened successfully, `false` otherwise. The method attempts to detect the encoding of the open file. It reads a few bytes at the current location and tries to detect the Byte Order Mark character `0xFFFE`. If found, the current position is advanced behind the detected character and the encoding property is set to one of the strings `UCS-2BE`, `UCS-2LE`, `UCS4-BE`, `UCS-4LE`, or `UTF-8`. If the marker character is not found, it checks for zero bytes at the current location and makes an assumption about one of the above formats (except UTF-8). If everything fails, the `encoding` property is set to the system encoding. **Note:** Be careful about opening a file more than once. The operating system usually permits you to do so, but if you start writing to the file using two different `File` objects, you can destroy your data.
mode	A string indicating the read/write mode. One of: `r`: (read) Opens for reading. If the file does not exist or cannot be found, the call fails. `w`: (write) Opens a file for writing. If the file exists, its contents are destroyed. If the file does not exist, creates a new, empty file. `e`: (edit) Opens an existing file for reading and writing.
type	Optional. In Mac OS, the type of a newly created file, a 4-character string. Ignored in Windows and UNIX.
creator	Optional. In Mac OS, the creator of a newly created file, a 4-character string. Ignored in Windows and UNIX.

openDlg *fileObj*.OpenDlg ([*prompt*][,*select*])	Opens the built-in platform-specific file-browsing dialog, in which the user can select an existing file to open. If the user clicks **OK**, returns a `File` or `Folder` object for the selected file or folder. If the user cancels, returns `null`. Differs from the class method `openDialog()` in that it presets the current folder to this `File` object's parent folder and the current file to this object's associated file.
prompt	Optional. A string containing the prompt text, if the dialog allows a prompt.
select	Optional. A file or files to be preselected when the dialog opens: ● In Windows, a string containing a comma-separated list of file types with descriptive text, to be displayed in the bottom of the dialog as a drop-down list from which the user can select which types of files to display. Each element starts with the descriptive text, followed by a colon and the file search masks for this text, separated by semicolons. For example, to display two choices, one labeled **Text Files** that allows selection of text files with extensions `.TXT` and `.DOC`, and the other labeled **All files** that allows selection of all files: `Text Files:*.TXT;*.DOC,All files:*` ● In Mac OS, a string containing the name of a function defined in the current JavaScript scope that takes a `File` object argument. The function is called for each file about to be displayed in the dialog, and the file is displayed only when the function returns `true`.
read *fileObj*.read ([*chars*])	Reads the contents of the file starting at the current position, and returns a string that contains up to the specified number of characters.
chars	Optional. An integer specifying the number of characters to read. By default, reads from the current position to the end of the file. If the file is encoded, multiple bytes might be read to create single Unicode characters.
readch *fileObj*.readch ()	Reads a single text character from the file at the current position, and returns it in a string. Line feeds are recognized as `CR`, `LF`, `CRLF`, or `LFCR` pairs. If the file is encoded, multiple bytes might be read to create single Unicode characters.
readln *fileObj*.readln ()	Reads a single line of text from the file at the current position, and returns it in a string. Line feeds are recognized as `CR`, `LF`, `CRLF`, or `LFCR` pairs. If the file is encoded, multiple bytes might be read to create single Unicode characters.
remove *fileObj*.remove ()	Deletes the file associated with this object from disk, immediately, without moving it to the system trash. Returns `true` if the file is deleted successfully. Does not resolve aliases; instead, deletes the referenced alias or shortcut file itself. **Note:** Cannot be undone. It is recommended that you prompt the user for permission before deleting.

`rename` *`fileObj`*`.rename (`*`newName`*`)`	Renames the associated file. Returns `true` on success. Does not resolve aliases, but renames the referenced alias or shortcut file itself.
`newName`	The new file or folder name, with no path.
`resolve` *`fileObj`*`.resolve ()`	If this object references an alias or shortcut, this method resolves that alias and returns a new `File` object that references the file-system element to which the alias resolves. Returns `null` if this object does not reference an alias, or if the alias cannot be resolved.
`saveDlg` *`fileObj`*`.saveDlg ([`*`prompt`*`][,`*`preset`*`])`	Opens the built-in platform-specific file-browsing dialog, in which the user can select an existing file location at which to save this file. If the user clicks **OK**, returns a `File` or `Folder` object for the selected file or folder. If the user cancels, returns `null`. Differs from the class method `saveDialog()` in that it presets the current folder to this `File` object's parent folder and the file to this object's associated file, and prompts the user to confirm before overwriting an existing file.
`prompt`	Optional. A string containing the prompt text, if the dialog allows a prompt.
`preset`	Optional. A file or files to be preselected when the dialog opens: ● In Windows, a string containing a comma-separated list of file types with descriptive text, to be displayed in the bottom of the dialog as a drop-down list from which the user can select which types of files to display. Each element starts with the descriptive text, followed by a colon and the file search masks for this text, separated by semicolons. For example, to display two choices, one labeled **Text Files** that allows selection of text files with extensions `.TXT` and `.DOC`, and the other labeled **All files** that allows selection of all files: `Text Files:*.TXT;*.DOC,All files:*` ● In Mac OS, a string containing the name of a function defined in the current JavaScript scope that takes a `File` object argument. The function is called for each file about to be displayed in the dialog, and the file is displayed only when the function returns `true`.
`seek` *`fileObj`*`.seek (`*`pos`*`, `*`mode`*`)`	Seeks to the specified position in the file, and returns `true` if the position was changed. The new position cannot be less than 0 or greater than the current file size.
`pos`	The new current position in the file as an offset in bytes from the start, current position, or end, depending on the `mode`.
`mode`	The seek mode, one of: 0: Seek to absolute position, where `pos=0` is the first byte of the file. 1: Seek relative to the current position. 2. Seek backward from the end of the file.
`tell` *`fileObj`*`.tell ()`	Returns the current position as a byte offset from the start of the file.

write *fileObj*.write (*text*[, *text*...]...)	Writes the specified text to the file at the current position. Returns `true` on success. For encoded files, writing a single Unicode character may write multiple bytes. **Note:** Be careful not to write to a file that is open in another application or object, as this can overwrite existing data.
text	One or more strings to write, which are concatenated to form a single string.
writeln *fileObj*.writeln (*text*[, *text*...]...)	Writes the specified text to the file at the current position, and appends a Line Feed sequence in the style specified by the `linefeed` property. Returns `true` on success. For encoded files, writing a single Unicode character may write multiple bytes. **Note:** Be careful not to write to a file that is open in another application or object, as this can overwrite existing data.
text	One or more strings to write, which are concatenated to form a single string.

Folder Object

Represents a file-system folder or directory in a platform-independent manner. All properties and methods resolve file system aliases automatically and act on the original file unless otherwise noted.

Folder object constructors

To create a `Folder` object, use the `Folder` function or the `new` operator. The constructor accepts full or partial path names, and returns the new object.

```
Folder ([path]); //can return a File object
new Folder ([path]); //always returns a Folder object
```

path	Optional. The absolute or relative path to the folder associated with this object, specified in URI format; see Specifying Paths. The value stored in the object is the absolute path. The path need not refer to an existing folder. If not supplied, a temporary name is generated. If the path refers to an existing file: ● The `Folder` function returns a `File` object instead of a `Folder` object. ● The `new` operator returns a `Folder` object for a nonexisting folder with the same name.

Folder class properties

These properties are available as static properties of the `Folder` class. It is not necessary to create an instance to access them.

appData	Folder	A `Folder` object for the folder that contains application data for all users. Read only. ● In Windows, the value of `%APPDATA%` (by default, `C:\Documents and Settings\All Users\Application Data`) ● In Mac OS, `/Library/Application Support`
commonFiles	Folder	A `Folder` object for the folder that contains files common to all programs. Read only. ● In Windows, the value of `%CommonProgramFiles%` (by default, `C:\Program Files\Common Files`) ● In Mac OS,`/Library/Application Support`
current	Folder	A `Folder` object for the current folder. Assign either a `Folder` object or a string containing the new path name to set the current folder.
fs	String	The name of the file system. Read only. One of `Windows`, `Macintosh`, or `Unix`.
myDocuments	Folder	A `Folder` object for the default document folder. Read only. ● In Windows, `C:\Documents and Settings\`*`username`*`\My Documents` ● In Mac OS, `~/Documents`
startup	Folder	A `Folder` object for the folder containing the executable image of the running application. Read only.

system	Folder	A Folder object for the folder containing the operating system files. Read only. ● In Windows, the value of %windir% (by default, C:\Windows) ● In Mac OS, /System
temp	Folder	A Folder object for the default folder for temporary files. Read only.
trash	Folder	A Folder object for the folder containing deleted items. Read only.
userData	Folder	A Folder object for the folder that contains application data for the current user. Read only. ● In Windows, the value of %APPDATA% (by default, C:\Documents and Settings*username*\Application Data) ● In Mac OS, ~/Library/Application Support

Folder class functions

These functions are available as a static methods of the Folder class. It is not necessary to create an instance in order to call them.

decode Folder.decode (*what*)	Decodes the specified string as required by RFC 2396 and returns the decoded string.
what	String. The encoded string to decode. All special characters must be encoded in UTF-8 and stored as escaped characters starting with the percent sign followed by two hexadecimal digits. For example, the string "my%20file" is decoded as "my file". Special characters are those with a numeric value greater than 127, except the following: / - _ . ! ~ * ' ( )
encode Folder.encode (*what*)	Encodes the specified string as required by RFC 2396 and returns the encoded string. All special characters are encoded in UTF-8 and stored as escaped characters starting with the percent sign followed by two hexadecimal digits. For example, the string "my file" is encoded as "my%20file". Special characters are those with a numeric value greater than 127, except the following: / - _ . ! ~ * ' ( )
what	String. The string to encode.
isEncodingAvailable File.isEncodingAvailable (*name*)	Returns true if your system supports the specified encoding, false otherwise.
name	String. The encoding name.

selectDialog `Folder.selectDialog ([prompt][,prompt])`	Opens the built-in platform-specific file-browsing dialog. If the user clicks **OK**, returns a `Folder` object for the selected folder. If the user cancels, returns `null`. Differs from the object method `selectDlg()` in that it does not preselect a folder.
prompt	Optional. A string containing the prompt text, if the dialog allows a prompt.
preset	Optional. A `Folder` object for a folder to be preselected when the dialog opens.

Folder object properties

These properties are available for `Folder` objects.

absoluteURI	String	The full path name for the referenced folder in URI notation. Read only.
alias	Boolean	When `true`, the object refers to a file system alias or shortcut. Read only.
created	Date	The creation date of the referenced folder, or `null` if the object does not refer to a folder on disk. Read only.
error	String	A message describing the last file system error; see File and Folder Error Messages. Setting this value clears any error message and resets the error bit for opened folders.
exists	Boolean	When `true`, the path name of this object refers to an existing folder. Read only.
fsName	String	The platform-specific name of the referenced folder as a full path name. Read only.
modified	Date	The date of the referenced folder's last modification, or `null` if the object does not refer to a folder on disk. Read only.
name	String	The name of the referenced folder without the path specification. Read only.
parent	Folder	The `Folder` object for the folder that contains this folder, or `null` if this object refers to the root folder of a volume. Read only.
path	String	The path portion of the absolute URI, or the empty string If the name does not have a path. Read only.
relativeURI	String	The path name for the referenced folder in URI notation, relative to the current folder. Read only.

Folder object functions

These functions are available for `Folder` objects.

`create` *`folderObj`*`.create ()`	Creates a folder at the location to which the path name points. Returns `true` if the folder was created successfully.
`execute` *`folderObj`*`.execute ()`	Opens this folder in the file browser (as if it had been double-clicked in a file browser). Returns `true` immediately if the folder was opened successfully.
`getFiles` *`folderObj`*`.getFiles ([`*`mask`*`])`	Returns an array of `File` and `Folder` objects for the contents of this folder, filtered by the supplied `mask`, or `null` if this object's referenced folder does not exist.
`mask`	Optional. A search mask for file names. A string that can contain question mark (`?`) and asterisk (`*`) wild cards. Default is `"*"`, which matches all file names. Can also be the name of a function that takes a `File` or `Folder` object as its argument. It is called for each file or folder found in the search; if it returns `true`, the object is added to the return array. **Note:** In Windows, all aliases end with the extension `.lnk`, which is stripped from the file name when found to preserve compatibility with other operating systems. You can search for all aliases by supplying the search mask `"*.lnk"`, but note that such code is not portable.
`getRelativeURI` *`folderObj`*`.getRelativeURI ([`*`basePath`*`])`	Returns a string containing the URI for this folder relative to the specified base path, in URI notation. If no base path is supplied, returns the URI relative to the path of the current folder.
`basePath`	Optional. A string containing the base path for the relative URI. Default is the current folder.
`remove` *`folderObj`*`.remove ()`	Deletes the empty folder associated with this object from disk, immediately, without moving it to the system trash. Returns `true` if the folder is deleted successfully. ● Folders must be empty before they can be deleted. ● Does not resolve aliases; instead, deletes the referenced alias or shortcut file itself. **Note:** Cannot be undone. It is recommended that you prompt the user for permission before deleting.
`rename` *`folderObj`*`.rename (`*`newName`*`)`	Renames the associated folder. Returns `true` on success. ● Does not resolve aliases; instead, renames the referenced alias or shortcut file itself.
`newName`	The new folder name, with no path.
`resolve` *`folderObj`*`.resolve ()`	If this object references an alias or shortcut, this method resolves that alias and returns a new `Folder` object that references the file-system element to which the alias resolves. Returns `null` if this object does not reference an alias, or if the alias cannot be resolved.

`selectDlg` *`folderObj`*`.selectDlg` `([`*`prompt`*`][,`*`preset`*`])`	Opens the built-in platform-specific file-browsing dialog. If the user clicks **OK**, returns a `File` or `Folder` object for the selected file or folder. If the user cancels, returns `null`. Differs from the class method `selectDialog()` in that it preselects this folder.
`prompt`	Optional. A string containing the prompt text, if the dialog allows a prompt.
`preset`	Optional. A `Folder` object for a folder to be preselected when the dialog opens.

File and Folder Error Messages

The following messages can be returned in the `error` property.

`File or folder does not exist`	The file or folder does not exist, but the parent folder exists.
`File or folder already exists`	The file or folder already exists.
`I/O device is not open`	An I/O operation was attempted on a file that was closed.
`Read past EOF`	Attempt to read beyond the end of a file.
`Conversion error`	The content of the file cannot be converted to Unicode.
`Partial multibyte character found`	The character encoding of the file data has errors.
`Permission denied`	The OS did not allow the attempted operation.
`Cannot change directory`	Cannot change the current folder.
`Cannot create`	Cannot create a folder.
`Cannot rename`	Cannot rename a file or folder.
`Cannot delete`	Cannot delete a file or folder.
`I/O error`	Unspecified I/O error.
`Cannot set size`	Setting the file size failed.
`Cannot open`	Opening of a file failed.
`Cannot close`	Closing a file failed.
`Read error`	Reading from a file failed.
`Write error`	Writing to a file failed.
`Cannot seek`	Seek failure.
`Cannot execute`	Unable to execute the specified file.

File and Folder Supported Encoding Names

The following list of names is a basic set of encoding names supported by the `File` object. Some of the character encoders are built in, while the operating system is queried for most of the other encoders. Depending on the language packs installed, some of the encodings may not be available. Names that refer to the same encoding are listed in one line. Underlines are replaced with dashes before matching an encoding name.

The `File` object processes an extended Unicode character with a value greater that 65535 as a Unicode surrogate pair (two characters in the range between 0xD700-0xDFFF).

Built-in encodings are:

```
US-ASCII, ASCII,ISO646-US,I SO-646.IRV:1991, ISO-IR-6,
ANSI-X3.4-1968,CP367,IBM367,US,ISO646.1991-IRV
UCS-2,UCS2, ISO-10646-UCS-2
UCS2LE,UCS-2LE,ISO-10646-UCS-2LE
UCS2BE,UCS-2BE,ISO-10646-UCS-2BE
UCS-4,UCS4, ISO-10646-UCS-4
UCS4LE,UCS-4LE,ISO-10646-UCS-4LE
UCS4BE,UCS-4BE,ISO-10646-UCS-4BE
UTF-8,UTF8,UNICODE-1-1-UTF-8,UNICODE-2-0-UTF-8,X-UNICODE-2-0-UTF-8
UTF16,UTF-16,ISO-10646-UTF-16
UTF16LE,UTF-16LE,ISO-10646-UTF-16LE
UTF16BE,UTF-16BE,ISO-10646-UTF-16BE
CP1252,WINDOWS-1252,MS-ANSI
ISO-8859-1,ISO-8859-1,ISO-8859-1:1987,ISO-IR-100,LATIN1
MACINTOSH,X-MAC-ROMAN
BINARY
```

The ASCII encoder raises errors for characters greater than 127, and the BINARY encoder simply converts between bytes and Unicode characters by using the lower 8 bits. The latter encoder is convenient for reading and writing binary data.

Additional encodings

In Windows, all encodings use code pages, which are assigned numeric values. The usual Western character set that Windows uses, for example, is the code page 1252. You can select Windows code pages by prepending the number of the code page with "CP" or "WINDOWS": for example, "CP1252" for the code page 1252. The `File` object has many other built-in encoding names that match predefined code page numbers. If a code page is not present, the encoding cannot be selected.

In Mac OS, you can select encoders by name rather than by code page number. The `File` object queries Mac OS directly for an encoder. As far as Mac OS character sets are identical with Windows code pages, Mac OS also knows the Windows code page numbers.

In UNIX, the number of available encoders depends on the installation of the `iconv` library.

Common encoding names

The following encoding names are implemented both in Windows and in Mac OS:

```
UTF-7,UTF7,UNICODE-1-1-UTF-7,X-UNICODE-2-0-UTF-7
ISO-8859-2,ISO-8859-2,ISO-8859-2:1987,ISO-IR-101,LATIN2
ISO-8859-3,ISO-8859-3,ISO-8859-3:1988,ISO-IR-109,LATIN3
ISO-8859-4,ISO-8859-4,ISO-8859-4:1988,ISO-IR-110,LATIN4,BALTIC
ISO-8859-5,ISO-8859-5,ISO-8859-5:1988,ISO-IR-144,CYRILLIC
ISO-8859-6,ISO-8859-6,ISO-8859-6:1987,ISO-IR-127,ECMA-114,ASMO-708,ARABIC
ISO-8859-7,ISO-8859-7,ISO-8859-7:1987,ISO-IR-126,ECMA-118,ELOT-928,GREEK8,GREEK
ISO-8859-8,ISO-8859-8,ISO-8859-8:1988,ISO-IR-138,HEBREW
```

```
ISO-8859-9,ISO-8859-9,ISO-8859-9:1989,ISO-IR-148,LATIN5,TURKISH
ISO-8859-10,ISO-8859-10,ISO-8859-10:1992,ISO-IR-157,LATIN6
ISO-8859-13,ISO-8859-13,ISO-IR-179,LATIN7
ISO-8859-14,ISO-8859-14,ISO-8859-14,ISO-8859-14:1998,ISO-IR-199,LATIN8
ISO-8859-15,ISO-8859-15,ISO-8859-15:1998,ISO-IR-203
ISO-8859-16,ISO-885,ISO-885,MS-EE
CP850,WINDOWS-850,IBM850
CP866,WINDOWS-866,IBM866
CP932,WINDOWS-932,SJIS,SHIFT-JIS,X-SJIS,X-MS-SJIS,MS-SJIS,MS-KANJI
CP936,WINDOWS-936,GBK,WINDOWS-936,GB2312,GB-2312-80,ISO-IR-58,CHINESE
CP949,WINDOWS-949,UHC,KSC-5601,KS-C-5601-1987,KS-C-5601-1989,ISO-IR-149,KOREAN
CP950,WINDOWS-950,BIG5,BIG-5,BIG-FIVE,BIGFIVE,CN-BIG5,X-X-BIG5
CP1251,WINDOWS-1251,MS-CYRL
CP1252,WINDOWS-1252,MS-ANSI
CP1253,WINDOWS-1253,MS-GREEK
CP1254,WINDOWS-1254,MS-TURK
CP1255,WINDOWS-1255,MS-HEBR
CP1256,WINDOWS-1256,MS-ARAB
CP1257,WINDOWS-1257,WINBALTRIM
CP1258,WINDOWS-1258
CP1361,WINDOWS-1361,JOHAB
EUC-JP,EUCJP,X-EUC-JP
EUC-KR,EUCKR,X-EUC-KR
HZ,HZ-GB-2312
X-MAC-JAPANESE
X-MAC-GREEK
X-MAC-CYRILLIC
X-MAC-LATIN
X-MAC-ICELANDIC
X-MAC-TURKISH
```

Additional Windows encoding names

```
CP437,IBM850,WINDOWS-437
CP709,WINDOWS-709,ASMO-449,BCONV4
EBCDIC
KOI-8R
KOI-8U
ISO-2022-JP
ISO-2022-KR
```

Additional Mac OS encoding names

These names are alias names for encodings that Mac OS might know.

```
TIS-620,TIS620,TIS620-0,TIS620.2529-1,TIS620.2533-0,TIS620.2533-1,ISO-IR-166
CP874,WINDOWS-874
JP,JIS-C6220-1969-RO,ISO646-JP,ISO-IR-14
JIS-X0201,JISX0201-1976,X0201
JIS-X0208,JIS-X0208-1983,JIS-X0208-1990,JIS0208,X0208,ISO-IR-87
JIS-X0212,JIS-X0212.1990-0,JIS-X0212-1990,X0212,ISO-IR-159
CN,GB-1988-80,ISO646-CN,ISO-IR-57
ISO-IR-16,CN-GB-ISOIR165
KSC-5601,KS-C-5601-1987,KS-C-5601-1989,ISO-IR-149
EUC-CN,EUCCN,GB2312,CN-GB
EUC-TW,EUCTW,X-EUC-TW
```

UNIX encodings

In UNIX, the `File` object looks for the presence of the `iconv` library, and uses whatever encoding it finds there. If you need a special encoding in UNIX, make sure that there is an `iconv` encoding module installed that converts between UTF-16 (the internal format that the `File` object uses) and the desired encoding.

8 ScriptUI Object Reference

Overview

ScriptUI is a component that works with the ExtendScript JavaScript interpreter to provide JavaScript programs with the ability to create and interact with user interface elements. It provides an object model for windows and UI control elements within an Adobe Creative Suite 2 application.

For an overview of the ScriptUI object model and a description of usage, see Chapter 5, “Using ScriptUI."

This chapter provides the details of the ScriptUI classes and objects with their properties, methods, and creation parameters.

- Window Class
- Window Object
- Control Objects
- Size and Location Objects
- LayoutManager Object
- MenuElement Object

Window Class

The `Window` class defines these static properties and functions which are available globally through reference to the class. Window instances created with `new Window()` do not have these properties and functions.

Window class properties

`coreVersion`	String	The internal core version number of the ScriptUI components. Read only.
`version`	String	The main version number of the ScriptUI components. Read only.

Window class functions

`alert` `Window.alert (`*`message`*`[, `*`title, errorIcon`*`])`	Displays the localizable *`message`* string in a user alert box that provides an **OK** button. For details, see the ExtendScript alert function. The alert dialog is not intended for lengthy messages. When the string argument is too long, the alert dialog truncates it.
`confirm` `Window.confirm (`*`message`*`[,`*`noAsDflt`*` ,`*`title`*` ])`	Displays the localizable *`message`* string in a self-sizing modal dialog box with **Yes** and **No** buttons. Returns `true` if the user clicks **Yes**, `false` if the user clicks **No**. For details, see the ExtendScript confirm function. The confirmation dialog can show longer messages than the alert and prompt dialogs, but if this string is too long, the dialog truncates it.

`find`
`Window.find (`*`resourceName`*`)`
`Window.find (`*`type, title`*`)`
	Finds and returns an existing window object, which can be a window already created by a script, or a windows created by the application (if the application supports this case).
	`resourceName`: A named resource that identifies a window that the application exposes to JavaScript. (Not supported in all ScriptUI implementations.)
	`type`: The window creation type, `dialog`, `palette`, or `window`. Used to distinguish between windows with the same title. If the type is unimportant, pass `null` or an empty string.
	`title`: The title of the window to find.
	If it finds an existing window, the method returns the corresponding JavaScript `window` object. If not, it returns `null`.

getResourceText `Window.getResourceText (textResource)`	Finds and returns a text string representation of the *textResource* from the host application's resource data. If no string resource matches the *textResource* name, the name is treated as literal text.
prompt `Window.prompt (message, preset[, title ])`	Displays a modal dialog that returns the user's text input. ● When the user clicks **OK** to dismiss the dialog, the method returns the text the user entered. ● When the user clicks the **Cancel** button, the method returns `null`. For details, see the ExtendScript prompt function.

Window Object

Window object constructor

To create a new `Window` object:

```
new Window (type [, title, bounds, {creation_properties}]);
```

type	The window type. The value is: `dialog`: Creates a modal dialog. `palette`: Creates a modeless dialog, also called a floating palette. `window`: Creates a simple window that can be used as a main window for an application
title	Optional. The window title. A localizable string.
bounds	Optional. The window's position and size.
creation_properties	Optional. An object that can contain any of these properties: `resizeable`: When `true`, the window can be resized by the user. Default is `false`. `closeButton`: When `true`, the title bar includes a button to close the window, if the platform and window type allow it. When `false`, it does not. Default is `true`. Not used for dialogs. `maximizeButton`: When `true`, the title bar includes a button to expand the window to its maximum size (typically, the entire screen), if the platform and window type allow it. When `false`, it does not. Default is `false` for type `palette`, `true` for type `window`. Not used for dialogs. `minimizeButton`: When `true`, the title bar includes a button to minimize or iconify the window, if the platform and window type allow it. When `false`, it does not. Default is `false` for type `palette`, `true` for type `window`. Main windows cannot have a minimize button in Mac OS. Not used for dialogs. `independent`: When `true`, a window of type `window` is independent of other application windows, and can be hidden behind them in Windows. In Mac OS, has no effect. Default is `false`.

Creates and returns a new `window` object, or `null` if window creation failed.

Window object properties

Window elements contain the following properties, in addition to those common to all ScriptUI elements:

defaultElement	Object	For a window of type dialog, the control to notify when a user types the ENTER key. By default, looks for a button whose name or text is "ok" (case disregarded).
cancelElement	Object	For a window of type dialog, the control to notify when a user types the ESC key in Windows, or the CMD . combination in Mac OS. By default, looks for a button whose name or text is "cancel" (case disregarded).
frameBounds	Bounds	A Bounds object for the boundaries of the Window's frame in screen coordinates. The frame consists of the title bar and borders that enclose the content region of a window, depending on the windowing system. Read only.
frameLocation	Point	A Point object for the location of the top left corner of the Window's frame. The same as [frameBounds.x, frameBounds.y]. Set this value to move the window frame to the specified location on the screen. The frameBounds value changes accordingly.
frameSize	Dimension	A Dimension object for the size and location of the Window's frame in screen coordinates. Read only.

Container properties

The following table shows properties that are available on `window` objects and container objects (controls of type `panel` and `group`).

alignChildren	String	Tells the layout manager how unlike-sized children of a container should be aligned within a column or row. Order of creation determines which children are at the top of a column or the left of a row; the earlier a child is created, the closer it is to the top or left of its column or row.
		If defined, `alignment` for a child element overrides the `alignChildren` setting for the parent container.
		Allowed values depend on the `orientation` value. For `orientation=row`:
		`top`
		`bottom`
		`center (default)`
		`fill`
		For `orientation=column`:
		`left`
		`right`
		`center (default)`
		`fill`
		For `orientation=stack`:
		`top`
		`bottom`
		`left`
		`right`
		`center (default)`
		`fill`
		Values are not case sensitive.

children	Array of Object	The collection of UI elements that have been added to this container (`window`, `panel`, `group`). An array indexed by number or by a string containing an element's `name`. The `length` property of this array is the number of child elements for container elements, and is zero for controls. Read only.
layout	LayoutManager	A LayoutManager Object for a container (`window`, `panel`, `group`). The first time a container object is made visible, ScriptUI invokes this layout manager by calling its `layout` function. Default is an instance of the `LayoutManager` class that is automatically created when the container element is created.
margins	Margins	A Margins object describing the number of pixels between the edges of this container and the outermost child elements. You can specify different margins for each edge of the container. The default value is based on the type of container, and is chosen to match the standard Adobe UI guidelines.

orientation	String	How elements are organized within this container. Interpreted by the layout manager for the container. The default LayoutManager Object accepts the (case-insensitive) values:
		`row`
		`column`
		`stack`
		The default orientation depends on the type of container. For `Window` and `Panel`, the default is `column`, and for `Group` the default is `row`.
		The allowed values for the container's `alignChildren` and its children's `alignment` properties depend on the orientation.

spacing	Number	The number of pixels separating one child element from its adjacent sibling element. Because each container holds only a single row or column of children, only a single spacing value is needed for a container. The default value is based on the type of container, and is chosen to match standard Adobe UI guidelines.

Window object functions

These functions are defined for `window` objects.

add `(type [, bounds, text, { creation_props> } ]);`	Creates and returns a new control or container object and adds it to the children of this window. Returns `null` if unable to create the object.
type	The control type. See Control types and creation parameters.
bounds	Optional. A bounds specification that describes the size and position of the new control or container, relative to its parent. See Bounds object for specification formats. If supplied, this value creates a new Bounds object which is assigned to the new object's `bounds` property.
text	Optional. String. Initial text to be displayed in the control as the title, label, or contents, depending on the control type. If supplied, this value is assigned to the new object's `text` property.
creation_props	Optional. Object. The properties of this object specify creation parameters, which are specific to each object type. See Control types and creation parameters.
center `windowObj.center ([window])`	Centers this window on the screen, or with respect to another specified window.
window	Optional. A Window Object.
close `windowObj.close ([result])`	Closes this window. If an onClose callback is defined for the window, calls that function before closing the window.
result	Optional. A number to be returned from the `show` method that invoked this window as a modal dialog.

hide *windowObj*.hide()	Hides this window. When a window is hidden, its children are also hidden, but when it is shown again, the children retain their own visibility states. For a modal dialog, closes the dialog and sets its result to 0.
notify *windowObj*.notify([*event*])	Sends a notification message, simulating the specified user interaction event. For example, to simulate a dialog being moved by a user: `myDlg.notify("onMove")`

event	Optional. The name of the window event handler to call. One of:
	`onClose`
	`onMove`
	`onMoving`
	`onResize`
	`onResizing`
	`onShow`

remove *windowObj*.remove(*index*) *windowObj*.remove(*text*) *windowObj*.remove(*child*)	Removes the specified child control from this window's `children` array. No error results if the child does not exist. Returns `undefined`.
index *text* *child*	The child control to remove, specified by 0-based index, `text` value, or as a `control` object.
show *windowObj*.show()	Shows this window, container, or control. If an onShow callback is defined for a window, calls that function before showing the window. When a window or container is hidden, its children are also hidden, but when it is shown again, the children retain their own visibility states. For a modal dialog, opens the dialog and does not return until the dialog is dismissed. If it is dismissed via the close method, this method returns any *result* value passed to that method. Otherwise, returns 0.

Window event-handling callbacks

The following callback functions can be defined to respond to events in windows. To respond to an event, define a function with the corresponding name in the `window` object.

Callback	Description
onClose	Called when a request is made to close the window, either by an explicit call to the `close` function or by a user action (clicking the OS-specific close icon in the title bar). The function is called before the window actually closes; it can return `false` to cancel the close operation.
onMove	Called when the window has been moved.
onMoving	Called while a window in being moved, each time the position changes. A handler can monitor the move operation.
onResize	Called when the window has been resized.

Callback	Description
`onResizing`	Called while a window is being resized, each time the height or width changes. A handler can monitor the resize operation.
`onShow`	Called when a request is made to open the window using the show method, before the window is made visible, but after automatic layout is complete. A handler can modify the results of the automatic layout.

Control Objects

Control object constructors

Use the `add` method to create new containers and controls. The `add` method is available on `window` and container (`panel` and `group`) objects. (See also add for dropdownlist and listbox controls.)

add `(type [, bounds, text, { creation_props> } ]);`	Creates and returns a new control or container object and adds it to the children of this window or container. Returns `null` if unable to create the object.
type	The control type. See Control types and creation parameters.
bounds	Optional. A bounds specification that describes the size and position of the new control or container, relative to its parent. See Bounds object for specification formats. If supplied, this value creates a new Bounds object which is assigned to the new object's `bounds` property.
text	Optional. String. Initial text to be displayed in the control as the title, label, or contents, depending on the control type. If supplied, this value is assigned to the new object's `text` property.
creation_props	Optional. Object. The properties of this object specify creation parameters, which are specific to each object type. See Control types and creation parameters.

Control types and creation parameters

The following type names can be used in string literals as the `type` specifier for the `add` method, available on `window` and container (`panel` and `group`) objects. The class names can used in resource specifications to define controls within a window or panel.

Type name	Class name	Description
button	Button	A pushbutton containing a text string. Calls the onClick callback if the control is clicked or if its notify method is called. To add to a window `w`: `w.add ("button" [, bounds, text ]);` *bounds*: Optional. The control's position and size. *text*: Optional. The text displayed in the control.
checkbox	Checkbox	A dual-state control showing a box with a checkmark when `value=true`, empty when `value=false`. Calls the onClick callback if the control is clicked or if its notify method is called. To add to a window `w`: `w.add ("checkbox" [, bounds, text ]);` *bounds*: Optional. The control's position and size. *text*: Optional. The text displayed in the control.

Type name	Class name	Description

dropdownlist	DropDownList	A drop-down list with zero or more items. Calls the onChange callback if the item selection is changed or if its notify method is called.
		
		To add to a window w:
		
		`w.add ("dropdown list", bounds [, items]`
		`[, {creation_properties}]);`
		bounds: The control's position and size.
		items: Optional. Supply this argument or the *creation_properties* argument, not both. An array of strings for the text of each list item. An item object is created for each item. An item with the text string "-" creates a separator item.
		creation_properties: Optional. Supply this argument or the *items* argument, not both. This form is most useful for elements defined using Resource Specifications. An object that contains the following property:
		items: An array of strings for the text of each list item. An item object is created for each item. An item with the text string "-" creates a separator item.

edittext	EditText	An edit text field that the user can change. Calls the onChange callback if the text is changed and the user types ENTER or the control loses focus, or if its notify method is called. Calls the onChanging callback when any change is made to the text. The textselection property contains currently selected text. To add to a window w: `w.add ("edittext" [, bounds, text, {creation_properties}]);` *bounds*: Optional. The control's position and size. *text*: Optional. The text displayed in the control. *creation_properties*: Optional. An object that contains any of the following properties: **multiline**: When false (the default), the control accepts a single line of text. When true, the control accepts multiple lines, in which case the text wraps within the width of the control. **readonly**: When false (the default), the control accepts text input. When true, the control does not accept input but only displays the contents of the text property. **noecho**: When false (the default), the control displays input text. When true, the control does not display input text (used for password input fields). **enterKeySignalsOnChange**: When false (the default), the control signals an onChange event when the editable text is changed and the control loses the keyboard focus (that is, the user tabs to another control, clicks outside the control, or types ENTER). When true, the control only signals an onChange event when the editable text is changed and the user types ENTER; other changes to the keyboard focus do not signal the event.

Type name	Class name	Description
group	Group	A container for other controls. Containers have additional properties that control the children; see Container properties. Hiding a group hides all its children. Making it visible makes visible those children that are not individually hidden. To add to a window w: `w.add ("group" [, bounds]);` *bounds*: Optional. The element's position and size.
iconbutton	IconButton	A pushbutton containing an icon. Calls the onClick callback if the control is clicked or if its notify method is called. To add to a window w: `w.add ("iconbutton" [, bounds, icon, {creation_properties}]);` *bounds*: Optional. The control's position and size. *icon*: Optional. The named resource for the icon or family of icons displayed in the button control, or a pathname or File Object for an image file. Images must be in PNG format. *creation_properties*: Optional. An object that contains the following property: **style**: A string for the visual style, one of: button: Has a visible border with a raised or 3D appearance. toolbutton: Has a flat appearance, appropriate for inclusion in a toolbar
image	Image	Displays an icon or image. To add to a window w: `w.add ("image" [, bounds, icon]);` *bounds*: Optional. The control's position and size. *icon*: Optional. The named resource for the icon or family of icons displayed in the image control, or a pathname or File Object for an image file. Images must be in PNG format.
item	ListItem	A choice item in a list box or drop-down list. The objects are created when items are specified on creation of the parent list object, or afterward using the list control's add method. ● Items in a drop-down list can be of type separator, in which case they cannot be selected, and are shown as a horizontal line. Item objects have these properties which are not found in other controls: index selected

Type name	Class name	Description
listbox	ListBox	A list box with zero or more items. Calls the onChange callback if the item selection is changed or if its notify method is called. To add to a window `w`: `w.add ("listbox", bounds [, items, {creation_properties}]);` *bounds*: Optional. The control's position and size. *items*: Optional. An array of strings for the text of each list item. An `item` object is created for each item. Supply this argument, or the `items` property in *creation_properties*, not both. *creation_properties*: Optional. An object that contains any of the following properties: **multiselect**: When `false` (the default), only one item can be selected. When `true`, multiple items can be selected. **items**: An array of strings for the text of each list item. An `item` object is created for each item. An item with the text string `"-"` creates a separator item. Supply this property, or the `items` argument, not both. This form is most useful for elements defined using Resource Specifications.
panel	Panel	A container for other types of controls, with an optional frame. Containers have additional properties that control the children; see Container properties. Hiding a panel hides all its children. Making it visible makes visible those children that are not individually hidden. To add to a window `w`: `w.add ("panel" [, bounds, text, {creation_properties} ]);` *bounds*: Optional. The element's position and size. A panel whose width is 0 appears as a vertical line. A panel whose height is 0 appears as a horizontal line. *text*: Optional. The text displayed in the border of the panel. *creation_properties*: Optional. An object that contains the following property: **borderStyle**: A string that specifies the appearance of the border drawn around the panel. One of `black`, `etched`, `gray`, `raised`, `sunken`. Default is `etched`.
progressbar	Progressbar	A horizontal rectangle that shows progress of an operation. All `progressbar` controls have a horizontal orientation. The `value` property contains the current position of the progress indicator; the default is 0. There is a `minvalue` property, but it is always 0; attempts to set it to a different value are silently ignored. To add to a window `w`: `w.add ("progressbar" [, bounds, value, maxvalue]);` *bounds*: Optional. The control's position and size. *value*: Optional. The initial position of the progress indicator. Default is 0. *maxvalue*: Optional. The maximum value that the `value` property can be set to. Default is 100.

Type name	Class name	Description
radiobutton	RadioButton	A dual-state control, grouped with other radiobuttons, of which only one can be in the selected state. Shows the selected state when `value=true`, empty when `value=false`. Calls the onClick callback if the control is clicked or if its notify method is called. All `radiobuttons` in a group must be created sequentially, with no intervening creation of other element types. Only one `radiobutton` in a group can be set at a time; setting a different `radiobutton` unsets the original one. To add to a window `w`: `w.add ("radiobutton" [, bounds, text ]);` *bounds*: Optional. The control's position and size. *text*: Optional. The text displayed in the control.
scrollbar	Scrollbar	A scrollbar with a draggable scroll indicator and stepper buttons to move the indicator. The `scrollbar` control has a horizontal orientation if the `width` is greater than the `height` at creation time, or vertical if its `height` is greater than its `width`. Calls the onChange callback after the position of the indicator is changed or if its notify method is called. Calls the onChanging callback repeatedly while the user is moving the indicator. The `value` property contains the current position of the scrollbar's indicator within the scrolling area, within the range of `minvalue` and `maxvalue`. The `stepdelta` property determines the scrolling unit for the up or down arrow; default is 1. The `jumpdelta` property determines the scrolling unit for a jump (as when the bar is clicked outside the indicator or arrows); default is 20% of the range between `minvalue` and `maxvalue`. To add to a window `w`: `w.add ("scrollbar" [, bounds, value, minvalue, maxvalue]);` *bounds*: Optional. The control's position and size. *value*: Optional. The initial position of the scroll indicator. Default is 0. *minvalue*: Optional. The minimum value that the `value` property can be set to. Default is 0. Together with *maxvalue*, defines the scrolling range. *maxvalue*: Optional. The maximum value that the `value` property can be set to. Default is 100. Together with *minvalue*, defines the scrolling range.

Type name	Class name	Description
`slider`	`Slider`	A slider with a moveable position indicator. All `slider` controls have a horizontal orientation. Calls the onChange callback after the position of the indicator is changed or if its notify method is called. Calls the onChanging callback repeatedly while the user is moving the indicator. The `value` property contains the current position of the indicator within the range of `minvalue` and `maxvalue`. To add to a window `w`: `w.add ("slider" [, bounds, value, minvalue, maxvalue]);` *bounds*: Optional. The control's position and size. *value*: Optional. The initial position of the indicator. Default is 0. *minvalue*: Optional. The minimum value that the `value` property can be set to. Default is 0. Together with *maxvalue*, defines the range. *maxvalue*: Optional. The maximum value that the `value` property can be set to. Default is 100. Together with *minvalue*, defines the range.
`statictext`	`StaticText`	A text field that the user cannot change. To add to a window `w`: `w.add ("statictext" [, bounds, text, {creation_properties}]);` *bounds*: Optional. The control's position and size. *text*: Optional. The text displayed in the control. *creation_properties*: Optional. An object that contains any of the following properties: **`multiline`**: When `false` (the default), the control displays a single line of text. When `true`, the control displays multiple lines, in which case the text wraps within the width of the control. **`scrolling`**: When `false` (the default), the displayed text cannot be scrolled. When `true`, the displayed text can be vertically scrolled using the UP ARROW and DOWN ARROW; this case implies `multiline=true`.

Control object properties

The following table shows the properties of all ScriptUI elements. Some values apply only to controls of particular types, as indicated.

<table>
<tr><td>active</td><td>Boolean</td><td>When true, the object is active, false otherwise. Set to true to make a given control or dialog active.
● A modal dialog that is visible is by definition the active dialog.
● An active palette is the front-most window.
● An active control is the one with focus—that is, the one that accepts keystrokes, or in the case of a Button, be selected when the user types a Return.</td></tr>
<tr><td>alignment</td><td>String</td><td>Applies to child elements of a container. If defined, this value overrides the alignChildren setting for the parent container. Allowed values depend on the orientation value of the parent container. For orientation=row:
top
bottom
center (default)
fill
For orientation=column:
left
right
center (default)
fill
For orientation=stack:
top
bottom
left
right
center (default)
fill
Values are not case sensitive.</td></tr>
<tr><td>bounds</td><td>Bounds</td><td>A Bounds object describing the boundaries of the element, in screen coordinates for window elements, and parent-relative coordinates for child elements. For windows, the bounds refer only to the window's content region.
Setting an element's size or location changes its bounds property, and vice-versa.</td></tr>
<tr><td>enabled</td><td>Boolean</td><td>When true, the control is enabled, meaning that it accepts input. When false, control elements do not accept input, and all types of elements have a grayed-out appearance.</td></tr>
<tr><td>helpTip</td><td>String</td><td>A brief help message (also called a tool tip) that is displayed in a small floating window when the mouse cursor hovers over a UI control element. Set to an empty string or null to remove help text.</td></tr>
</table>

icon	String or File	The name of an icon resource or the pathname or File Object for a file that contains a platform-specific icon image in PNG format. ● For an IconButton, the icon appears as the content of the button. ● For a ListItem, the icon is displayed to the left of the text. ● For an Image, the icon is the entire content of the image element.
index	Number	For ListItem objects only. The index of this item in the items collection of its parent list control. Read only.
items	Array of Object	For a list object (listbox or dropdown list), a collection of ListItem objects for the items in the list. Access by 0-based index. To obtain the number of items in the list, use items.length. Read only.
itemSize	Dimension	For a list object (listbox or dropdown list), a Dimension object describing the width and height in pixels of each item in the list. Used by auto-layout to determine the preferredSize of the list, if not otherwise specified. If not set explicitly, the size of each item is set to match the largest height and width among all items in the list
jumpdelta	Number	The amount to increment or decrement a scrollbar indicator's position when the user clicks ahead or behind the moveable element. Default is 20% of the range between the maxvalue and minvalue property values.
justify	String	The justification of text in static text and edit text controls. One of: left (default) center right **Note:** Justification only works if the value is set before the window containing the control is displayed for the first time.
location	Point	A Point object describing the location of the element as an array, [x, y], representing the coordinates of the upper left corner of the element. These are screen coordinates for window elements, and parent-relative coordinates for other elements. The location is defined as [bounds.x, bounds.y]. Setting an element's location changes its bounds property, and vice-versa. By default, location is undefined.
maxvalue	Number	The maximum value that the value property can have. If maxvalue is reset less than value, value is reset to maxvalue. If maxvalue is reset less than minvalue, minvalue is reset to maxvalue.

minvalue	Number	The minimum value that the `value` property can have. If `minvalue` is reset greater than `value`, `value` is reset to `minvalue`. If `minvalue` is reset greater than `maxvalue`, `maxvalue` is reset to `minvalue`.
parent	Object	The parent object of a UI object, a `window`, `panel` or `group`, or `null` for `window` objects. Read only.
preferredSize	Dimension	A Dimension object used by layout managers to determine the best size for each element. If not explicitly set by a script, value is established by the UI framework in which ScriptUI is employed, and is based on such attributes of the element as its text, font, font size, icon size, and other UI framework-specific attributes. A script can explicitly set `preferredSize` before the layout manager is invoked in order to establish an element size other than the default.
properties	Object	An object that contains one or more creation properties of the element (properties used only when the element is created).
selected	Boolean	For `ListItem` objects only. When `true`, the item is part of the `selection` for its parent list. When `false`, the item is not selected. Set to true to select this item in a single-selection list, or to add it to the to the selection array for a multi-selection list.
selection	ListItem, Array of ListItem	For a list object (`listbox` or `dropdown` list), the currently selected `ListItem` object for a single-selection list, or an array of `ListItem` objects for current selection in a multi-selection list. Setting this value causes the selected item to be highlighted and to be scrolled into view if necessary. You can set the value using the index of an item or an array of indices, rather than object references. If set to an index value that is out of range, the operation is ignored. When set with index values, the property still returns object references. ● If you set the value to an array for a single-selection list, only the first item in the array is selected. ● If you set the value to a single item for a multi-selection list, that item is added to the current selection. If no items are selected, the value is `null`. Set to `null` to deselect all items.
size	Dimension	A Dimension object that defines the actual dimensions of an element. Initially `undefined`, and unless explicitly set by a script, it is defined by a `LayoutManager`. A script can explicitly set size before the layout manager is invoked to establish an element size other than the `preferredSize` or the default size. Defined as `[bounds.width, bounds.height]`. Setting an element's `size` changes its `bounds` property, and vice-versa.
stepdelta	Number	The amount by which to increment or decrement a `Scrollbar` element's position when the user clicks a stepper button.

text	String	The title, label, or displayed text. Ignored for certain window types. For controls, the meaning depends on the control type. Buttons use the `text` as a label, for example, while edit fields use the `text` to access the content. This is a localizable string: see Localization in ScriptUI Objects
textselection	String	The currently selected text in a control that displays text, or the empty string if there is no text selected. Setting the value replaces the current text selection and modifies the value of the `text` property. If there is no current selection, inserts the new value into the `text` string at the current insertion point. The `textselection` value is reset to an empty string after it modifies the `text` value. **Note:** Setting the `textselection` property before the `edittext` control's parent `Window` exists is an undefined operation.
type	String	Contains the type name of the element, as specified on creation. ● For `window` objects, one of the type names `window`, `palette`, or `dialog`. ● For controls, the type of the control, as specified in the `add` method that created it. Read only.
value	Boolean	For a checkbox or radiobutton, `true` if the control is in the selected or set state, `false` if it is not.
value	Number	For a scrollbar or slider, the current position of the indicator. If set to a value outside the range specified by `minvalue` and `maxvalue`, it is automatically reset to the closest boundary.
visible	Boolean	When `true`, the element is shown, when `false` it is hidden. When a container is hidden, its children are also hidden, but they retain their own visibility values, and are shown or hidden accordingly when the parent is next shown.

Control object functions

The following table shows the methods defined for each element type, and for specific control types as indicated.

add *listObj*.add (*type*, *text*[, *index*])	For list objects (listbox or dropdown list) only. Adds an item to the items array at the given index. Returns the item control object for type=item, or null for type=separator.
type	The type of item to add. One of: item: A basic, selectable item with a text label. separator: A separator. For dropdownlist controls only. In this case, the *text* value is ignored, and the method returns null.
text	The localizable text label for the item.
index	Optional. The index into the current item list after which this item is inserted. If not supplied, or greater than the current list length, the new item is added at the end.
find *listObj*.find(*text*)	For list objects (listbox or dropdown list) only. Looks in this object's items array for an item object with the given text value. Returns the item object if found; otherwise, returns null.
text	The text of the item to find.
hide *controlObj*.hide()	Hides this container or control. When a window or container is hidden, its children are also hidden, but when it is shown again, the children retain their own visibility states.
notify *controlObj*.notify([*event*])	Sends a notification message, simulating the specified user interaction event.
event	Optional. The name of the control event handler to call. One of: onClick onChange onChanging By default, simulates the onChange event for an edittext control, an onClick event for controls that support that event.
remove *containerObj*.remove(*index*) *containerObj*.remove(*text*) *containerObj*.remove(*child*)	For containers (panel, group), removes the specified child control from the container's children array. For list objects (listbox or dropdown list) only, removes the specified item from this object's items array. No error results if the item does not exist.
index *text* *child*	The item or child to remove, specified by 0-based index, text value, or as a control object.
removeAll *listObj*.removeAll()	For list objects (listbox or dropdown list) only. Removes all items from the object's items array.
show *controlObj*.show()	Shows this container or control. When a window or container is hidden, its children are also hidden, but when it is shown again, the children retain their own visibility states.

toString *listItemObj*.toString()	For item controls only. Returns the value of this item's text property as a string.
valueOf *listItemObj*.valueOf()	For item controls only. Returns the index number of this item in the parent list's items array.

Control event-handling callbacks

The following events are signalled in certain types of controls. To handle the event, define a function with the corresponding name in the control object.

onClick	Called when the user clicks one of the following control types: button checkbox iconbutton radiobutton
onChange	Called when the user finishes making a change in one of the following control types: dropdownlist edittext listbox scrollbar slider ● For an edittext control, called only when the change is complete—that is, when focus moves to another control, or the user types ENTER. The exact behavior depends on the creation parameter enterKeySignalsOnChange; see the edittext description. ● For a slider or scrollbar, called when the user has finished dragging the position marker or has clicked the control.
onChanging	Called for each incremental change in one of the following control types: edittext scrollbar slider ● For an edittext control, called for each keypress while the control has focus. ● For a slider or scrollbar, called for any motion of the position marker.

Size and Location Objects

ScriptUI defines objects to represent the complex values of properties that place and size windows and UI elements. These objects cannot be created directly, but are created when you set the corresponding property. That property then returns that object. For example, the `bounds` property returns a `Bounds` object.

You can set these properties as objects, strings, or arrays.

- `e.`*`prop`*` = Object`: The object must contain the set of properties defined for this type, as shown in the table below. The properties have integer values.
- `e.`*`prop`*` = String`: The string must be an executable JavaScript inline object declaration, conforming to the same object description.
- `e.`*`prop`*` = Array`: The array must have integer coordinate values in the order defined for this type, as shown in the table below. For example:

The following examples show equivalent ways of placing a 380 by 390 pixel window near the upper left corner of the screen:

```
var dlg = new Window('dialog', 'Alert Box Builder');
dlg.bounds = {x:100, y:100, width:380, height:390}; //object
dlg.bounds = {left:100, top:100, right:480, bottom:490}; //object
dlg.bounds = "x:100, y:100, width:380, height:390"; //string
dlg.bounds = "left:100, top:100, right:480, bottom:490"; //string
dlg.bounds = [100,100,480,490]; //array
```

You can access the resulting object as an array with values in the order defined for the type, or as an object with the properties supported for the type.

The following table shows the property-value object types, the element properties that create and contain them, and their array and object-property formats.

`Bounds`	Defines the boundaries of a window within the screen's coordinate space, or of a UI element within the container's coordinate space. Contains an array, `[left, top, right, bottom]`, that defines the coordinates of the upper left and lower right corners of the element. A `Bounds` object is created when you set an element's `bounds` property, and this property returns a `Bounds` object. ● An object must contain properties named `left`, `top`, `right`, `bottom`, or `x`, `y`, `width`, `height`. ● An array must have values in the order `[left, top, right, bottom]`.
`Dimension`	Defines the size of a `window` or UI element. Contains an array, `[width, height]`, that defines the element's size in pixels. A `Dimension` object is created when you set an element's `size` or `preferredSize` property. ● An object must contain properties named `width` and `height`. ● An array must have values in the order `[width, height]`.

Margins	Defines the number of pixels between the edges of a container and its outermost child elements. Contains an array [left, top, right, bottom] whose elements define the margins between the left edge of a container and its leftmost child element, and so on. A Margins object is created when you set an element's margins property. ● An object must contain properties named left, top, right, and bottom. ● An array must have values in the order [left, top, right, bottom]. You can also set the margins property to a number; all of the array values are then set to this number.
Point	Defines the location of a window or UI element. Contains an array, [x, y], whose values represent the origin point of the element as horizontal and vertical pixel offsets from the origin of the element's coordinate space. A Point object is created when you set an element's location property. ● An object must contain properties named x and y. ● An array must have values in the order [x, y].

LayoutManager Object

Controls the automatic layout behavior for a window or container. The subclass AutoLayoutManager implements the default automatic layout behavior.

AutoLayoutManager object constructor

Create an instance of the `AutoLayoutManager` class with the `new` operator:

```
myWin.layout = new AutoLayoutManager(myWin);
```

An instance is automatically created when you create a `window` or container (`group` or `panel`) object, and referenced by the container's layout property. This instance implements the default layout behavior unless you override it.

AutoLayoutManager object properties

The default object has no predefined properties, but a script can assign arbitrary properties to an object it creates, to store data needed by the script-defined layout algorithm.

AutoLayoutManager object functions

`layout` *`win`*`.layout.layout (`*`recalculate`*`)`	Invokes the automatic layout behavior for the managed container. Adjusts sizes and positions of the child elements of this window or container according to the placement and alignment property values in the parent and children. Invoked automatically the first time the window is displayed. Thereafter, the script must invoke it explicitly to change the layout in case of changes in the size or position of the parent or children.
`recalculate`	Optional. When `true`, forces the layout manager to recalculate the container size for this and any child containers. Default is `false`.

MenuElement Object

The `MenuElement` class is used to represent application menu bars, their menus and submenus, and individual items or commands. Each application creates `menuElement` instances for each of the existing menu elements, and you can create additional instances to extend the existing menus.

Each `menuElement` object has unique identifier. Existing menu elements that can be extended have predefined identifiers, listed in Bridge menu and command identifiers. Not all existing menu elements can be extended. You can only add a new menu or command before or after an existing menu or command, which you must specify using the predefined unique identifier.

The menu, submenu, and command identifier names do not necessarily match the display names. Menu identifiers are case sensitive. They are not displayed and are never localized. When a script creates a new menu or command, you should assign a descriptive unique identifier. If you do not, one is generated using a numeric value.

The display text of a new menu element can be localized by specifying it with the Global localize function. See Localizing ExtendScript Strings.

Menu separators are not independent elements, but can be inserted before or after an element that you add to a menu. The separator is specified as part of the location string on creation; see Creating new menu elements below.

MenuElement class functions

The `MenuElement` class defines these static functions that you can use to extend and work with existing menu elements.

create `MenuElement.create (type, text, location[, id]);`	Adds a new menu to a menu bar, a new submenu to an existing menu, or a new command to an existing menu or submenu. Returns the new `menuElement` object. See examples below.
type	The type of menu element, one of: `menu`: a menu or submenu `command`: a menu item
text	The localizable string that is displayed as the label text. Script-created menu and menu commands cannot have keyboard shortcuts or icons.

location	A string describing the location of the new menu element, with respect to existing menu elements. This can take one of the following forms: `before` *`identifier`*: Create the new element before the given menu element. `after` *`identifier`*: Create the new element before the given menu element. `at the end of` *`identifier`*: Append the new element to the given menu. The identifier must be for a menu, not a command item. `at the beginning of` *`identifier`*: Create the new element as the first item in the given menu. The identifier must be for a menu, not a command item. To insert a separator before or after the new element, specify a dash (-) at the beginning or end of the location string. For example, this value draws separators before and after the new element, which is added after the Find submenu in the Edit menu: `-after /bridge/edit/find-` A string that does not conform to these rules causes a run-time error.
id	The unique identifier for this element. Optional. ● If the ID of an existing menu or submenu is supplied, the call returns that menu or submenu object. ● If the ID of an existing menu command is supplied, the call causes a JavaScript error. ● If not supplied, the call generates a numeric value, which can be found in the `id` property of the returned menu object.
`find` `MenuElement.find (`*`id`*`)`	Finds and returns the `menuElement` object for the specified menu or menu item, or `null` if no such element is found.
id	String. The unique identifier for the menu element to find.

➤ ***Example***

This example checks to see whether a specific menu item already exists to avoid an error if the script is executed a second time.

```
var menu = MenuElement.find ('myMenuId');
if (menu = null) //element does not yet exist
// add menu element
```

`remove` `MenuElement.remove (`*`id`*`)`	Removes a script-defined menu or menu item. Returns `undefined`.
id	String. The unique identifier for the menu element to remove.

Creating new menu elements

These examples illustrate the creation of new menus and menu items.

➤ Example: Adding a menu and command to the menu bar

This example adds a new menu to the menu bar, after the **Help** menu. It adds one command to that menu, labeled "Alert", and assigns it an `onSelect` callback that displays an alert dialog when the item is clicked.

```
newMenu = new MenuElement( "menu", "MyMenu", "after Help", "myMenu" );
alertCommand = new MenuElement( "command", "Alert", "at the end of myMenu",
   "myAlert" );
```

```
alertCommand.onSelect = function () { Window.alert("Hi."); }
```

➤ Example: Adding a command to a context menu

This example adds a "Count Children" command to the context menu for folder thumbnails, and assigns it an `onSelect` callback that counts and displays the number of child nodes in that folder.

The `onSelect` callback assumes that the thumbnail is for a folder, so the example makes sure it cannot be called for a thumbnail that does not represent a folder. To do this, the `onDisplay` callback of the new element (called each time the menu is displayed) enables the command only when the currently selected thumbnail is for a folder.

If multiple thumbnails are selected when the user invokes the context menu, the new command is enabled if the first one is a folder. In this case, selecting the command reports the number of items in that folder.

```
var cntCommand = new MenuElement("command", "Count Children",
      "at the end of Thumbnail", "myCount");
cntCommand.onSelect = function(m) {
   try {
   // get the thumbnail associated with the context menu
      var tn = app.document.selections[0];
   // display the number of direct descendants
      Window.alert("Number of direct descendants: " + tn.children.length);
   } catch(error) { Window.alert(error); }
};
cntCommand.onDisplay = function(m) {
   try {
      var tn = app.document.selections[0];//check the first selected node
      if (tn.container) //is it for a folder?
         m.enabled = true; // yes, enable the command
      else
         m.enabled = false; // no, disable the command
   } catch(error) { Window.alert(error); }
};
```

MenuElement object properties

altDown	Boolean	When true, the ALT modifier key was pressed when the item was selected. Read only.
checked	Boolean	When `true`, the command is selected. A check mark appears next to the label. When `false`, the item is not selected, and no check mark is shown. Read/write.
cmdDown	Boolean	When true, the COMMAND modifier key was pressed when the item was selected. Read only.
ctrlDown	Boolean	When true, the CONTROL modifier key was pressed when the item was selected. Read only.
enabled	Boolean	When `true`, the menu or command is selectable. When `false`, it is grayed out and cannot be selected. Read/write.

id	String	A unique identifier. Read only. Identifiers take the form: `/app/menu/submenu/command` They are not localized, and are case sensitive.
onDisplay	Function	The callback function that is called when the application is about to display this menu or menu item. The function takes no arguments, and returns nothing. It can change the `enabled` and `checked` properties according to the state of the application.
optionDown	Boolean	When true, the OPTION modifier key was pressed when the item was selected. Read only.

onSelect	Function	The callback function that is called when the user selects the menu or menu item. The function takes no arguments, and returns nothing. It implements the behavior of a menu item.
		The callback can check this object's properties to respond to the following modifier keys:
		`if (this.ShiftDown)`
		`// Shift key pressed`
		`if (this.altDown)`
		`// Alt key pressed`
		`if (this.ctrlDown)`
		`// Control key pressed`
		`if (this.cmdDown)`
		`// Command key pressed`
		`if (this.optionDown)`
		`// Option key pressed`

shiftDown	Boolean	When true, the SHIFT modifier key was pressed when the item was selected. Read only.
text	String	The displayed label text, a localizable string. Read only.
type	String	The type of menu element, one of: `menu`: a menu or submenu `command`: a menu item Read only.

Bridge menu and command identifiers

These unique identifiers are predefined for Bridge menus that can be extended.

Bridge menu identifiers

These tables list unique identifiers for the top-level menus in Adobe Bridge

Menubar menus	Menu ID
Bridge (Mac OS only)	(not available)
File	`File`
Edit	`Edit`

Tools	`Tools`
Label	`Labels`
View	`View`
Window	`Window`
Help	`Help`
Context menus	**Menu ID**
thumbnail context	`Thumbnail`
Flyout menus	**Menu ID**
Folders tab flyout	`FoldersTab`
Keywords tab flyout	`KeywordsTab`
Metadata tab flyout	`MetadataTab`
Flyout menu submenus	**Menu ID**
Metadata flyout > AppendMetadata	`Bridge/Submenu/AppendMetadata`
Metadata flyout > ReplaceMetadata	`Bridge/Submenu/ReplaceMetadata`
Note: The commands in flyout menus are not available to scripts.	

Bridge submenu and command identifiers

These tables list unique identifiers for submenus and commands in the Adobe Bridge menus.

Bridge menu commands (Mac OS only)

Submenus/commands	Menu ID
About Bridge	`mondo/command/about`
Preferences	`Prefs`
Quit Bridge	`mondo/command/quit`

File menu commands

Submenus/commands	Menu ID
New Window	`mondo/command/new`
New Folder	`NewFolder`
Open	`Open`
Open with >	`submenu/OpenWith`
Open with > [*installed application*]	(not available)
Open in Camera Raw	`OpenInCameraRaw`

Eject	`Eject`
Close Window	`mondo/command/close`
Send to Recycle Bin	`MoveToTrash`
Return to	`ReturnToApplication`
Reveal in Explorer/Finder	`Reveal`
Reveal in Bridge	`RevealInBridge`
Place >	`submenu/Place`
Add To Favorites	`AddToFavorites`
File Info...	`FileInfo`
Versions...	`Versions`
Alternates...	`Alternates`
Exit	`mondo/command/quit`

Edit menu commands

Submenus/commands	Menu ID
Undo	`mondo/command/undo`
Cut	`mondo/command/cut`
Copy	`mondo/command/copy`
Paste	`mondo/command/paste`
Duplicate	`Duplicate`
Select All	`mondo/command/selectAll`
Select Labeled	`SelectLabeled`
Select Unlabeled	`SelectUnlabeled`
Invert Selection	`InvertSelection`
Deselect All	`mondo/command/selectNone`
Find...	`Search`
Camera Raw Settings...	`submenu/CameraRaw`
Apply Camera Raw Settings >	`ApplyCameraRaw`
Apply Camera Raw Settings > Camera Default	`CRDefault`
Apply Camera Raw Settings > Previous Conversion	`CRPrevious`
Apply Camera Raw Settings > Copy Camera Raw Settings	`CRCopy`
Apply Camera Raw Settings > Paste Camera Raw Settings	`CRPaste`

Apply Camera Raw Settings > Clear Settings	`CRClear`
Rotate 180°	`Rotate180`
Rotate 90° Clockwise	`Rotate90CW`
Rotate 90° Counterclockwise	`Rotate90CCW`
Creative Suite Color Settings...	`SharedSettings`
Camera Raw Preferences...	`CRPreferences`
Preferences...	`Prefs`

Tools menu commands

Submenus/commands	Menu ID
Batch Rename...	`BatchRename`
Version Cue >	`submenu/VersionCue`
Version Cue > Synchronize	`Synchronize`
Version Cue > Mark In Use	`CheckOut`
Version Cue > Save a Version	(not available)
Version Cue > Revert to Last Version	`RevertToProject`
Version Cue > Make Alternates	`CreateAlternateGroup`
Version Cue > New Project	`NewProject`
Version Cue > Connect to...	`ConnectTo`
Version Cue > Edit Properties...	`EditProperties`
Cache >	`submenu/Cache`
Cache > Build Cache for Subfolders	`BuildSubCaches`
Cache > Purge Cache for This Folder	`PurgeCache`
Cache > Purge Entire Cache	`PurgeAllCaches`
Cache > Export Cache	`ExportCache`
Append Metadata >	`Bridge/Submenu/AppendMetadata`
[*templates*]	(not available)
Replace Metadata >	`Bridge/Submenu/ReplaceMetadata`
[*templates*]	(not available)

Label menu commands

Submenus/commands	Menu ID
Rating	(not available)

No Rating	NoDot
*	OneDot
**	TwoDots
***	ThreeDots
****	FourDots
*****	FiveDots
Decrease Rating	RemoveDot
Increase Rating	AddDot
Label	(not available)
No Label	NoLabel
Red	Red
Yellow	Yellow
Green	Green
Blue	Blue
Purple	Purple

View menu commands

Submenus/commands	Menu ID
Compact Mode	ToggleCompactMode
Slide Show...	SlideShow
As Thumbnails	View/Thumbnail
As Details	View/Details
As Versions Alternates	View/Versions
As Filmstrip	View/Filmstrip
Favorites Panel	FavoritesTab
Folders Panel	FoldersTab
Preview Panel	PreviewTab
Metadata Panel	MetadataTab
Keywords Panel	KeywordsTab
Sort >	submenu/Sort
Sort > Ascending Order	Ascending
Sort > By File Name	SortFileName

Sort > By Document Kind	SortFileType
Sort > By Date Created	SortDateCreated
Sort > By Date File Modified	SortDateModified
Sort > By File Size	SortFileType
Sort > By Dimensions	SortDimensions
Sort > By Resolution	SortResolution
Sort > By Color Profile	SortColorProfile
Sort > By Copyright	SortCopyright
Sort > By Label	SortByLabel
Sort > By Rating	SortRating
Sort > By Purchase State	SortPurchaseState
Sort > By Version Cue Status	SortUseState
Sort > Manually	SortManually
Show Thumbnail Only	ShowThumbnailOnly
Show Hidden Files	ShowHidden
Show Folders	ShowFolders
Show All Files	FilterNoFiles
Show Graphic Files Only	FilterGraphicFiles
Show Camera Raw Files Only	FilterCameraRawFiles
Show Vector Files Only	FilterVectorFiles
Refresh	Refresh

Window menu commands

Submenus/commands	Menu ID
Workspace >	submenu/Workspace
Workspace > Save Workspace	SaveWorkspace
Workspace > Delete Workspace	DeleteWorkspace
Workspace > Reset to Default Workspace	(not available)
Workspace > Lightbox	(not available)
Workspace > File Navigator	(not available)
Workspace > Metadata Focus	(not available)
Workspace > Filmstrip Focus	(not available)

Help menu commands

Submenus/command	Menu ID
Bridge Help...	`mondo/command/help`
VersionCue Help...	`VersionCueHelp`
Updates...	`Updates`
About Bridge...	`mondo/command/about`

Context menu commands

Thumbnail context menu in Favorites tab commands	**Menu ID**
Remove From Favorites	`Bridge/ContextMenu/Keyword/Delete`
Thumbnail context menu in Folders tab commands	**Menu ID**
Send to Recycle Bin	`Bridge/ContextMenu/Folders/Delete`
Reveal in Explorer/Finder	`Bridge/ContextMenu/Folders/Reveal`
Add to Favorites	`Bridge/ContextMenu/Folders/AddToFavorites`
Thumbnail context menu in Content pane (folders) submenus/commands	**Menu ID**
Open	`Thumbnail/Open`
Open with >	`submenu/OpenWith`
Reveal in Explorer/Finder	`Thumbnail/RevealLocation`
Add to Favorites	`Thumbnail/AddToFavorites`
Send to Recycle Bin	`Thumbnail/Remove`
Label >	`submenu/Label`
Label > No Label	(not available)
Label > *label strings*	(not available)
Thumbnail context menu in Content pane (files) submenus/commands	**Menu ID**
Open	`Thumbnail/Open`
Open With >	`subment/OpenWith`
Reveal in Explorer/Finder	`Thumbnail/RevealLocation`
Send to Recycle Bin	`Thumbnail/Remove`
File Info...	`Thumbnail/FileInfo`
Label >	`submenu/Label`
Label > No Label	(not available)

Label > *label strings*	(not available)
Thumbnail context menu in Content pane (images) additional commands	**Menu ID**
Rotate 180°	`Thumbnail/Rotate180`
Rotate 90° Clockwise	`Thumbnail/RotateCW`
Rotate 90° Counterclockwise	`Thumbnail/RotateCCW`
Thumbnail context menu in Content pane (Version Cue nodes) additional command	**Menu ID**
Versions...	`Thumbnail/Versions`
Keywords context menu commands	**Menu ID**
New Keyword	`Bridge/ContextMenu/Keyword/NewKey`
New Keyword Set	`Bridge/ContextMenu/Keyword/NewSet`
Rename	`Bridge/ContextMenu/Keyword/Rename`
Delete	`Bridge/ContextMenu/Keyword/DeleteNode`
Find...	`Bridge/ContextMenu/Keyword/Search`

9 Interapplication Communication with Scripts

Scripts written for any Adobe Creative Suite 2 application can communicate with other Creative Suite 2 applications in three ways:

- Cross-DOM Functions

  This limited set of basic functions is common across all Adobe Creative Suite 2 applications, and allows your script to, for example, open or print files in other applications, simply by calling the `open` or `print` function for that application. Cross-DOM API Reference provides reference details for the functions of the Cross-DOM.

- Cross-DOM API Reference

  Each Adobe Creative Suite 2 application exports a set of functions to provide a selected set of application-specific functionality. For example, a Bridge script can request a photo merge in photoshop by calling `photoshop.photomerge(`*`files`*`)`. The set of functions available for each application varies widely.

- Communicating Through Interapplication Messages

  The interapplication message framework is an application programming interface (API) that allows extensive control over communication between applications. The API allows you to send messages to other applications and receive results, and to receive messages sent by other applications and return results. Typically the data passed between applications are JavaScript scripts. However, the messaging framework is extensible. It allows you to define different types of data to send between applications, and to specify how they are handled.

  Interapplication Message API Reference provides complete reference details.

Any application that supports any of these techniques is said to be *messaging enabled*. All Adobe Creative Suite 2 applications are messaging enabled. A set of Sample Workflow Automation Scripts is provided with Adobe Creative Suite 2, which demonstrate how scripts can be used to create a workflow that takes advantage of functionality in different applications.

When calling external functions or exchanging messages, you must identify particular applications using *namespace specifiers*. A specifier consists of a specific name string (such as `photoshop`), and optional additions that identify a particular release or locale version. Application specifiers are used occasionally in other contexts as well. For details of the syntax, see Application and Namespace Specifiers.

Regardless of which method you use to perform interapplication communication, you must place your script in a location where the application you want to run it can see it. There are different locations for the startup scripts of the applications themselves, and for scripts provided by developers.

Because all Adobe Creative Suite 2 applications look in the same locations for scripts to run, the scripts themselves must be explicit about which application they are meant for. A script should check that all applications it needs to communicate with are installed with the correct version, and that any other applications that might be installed do not run the script. For details, see Script Locations and Checking Application Installation.

Cross-DOM Functions

The Cross-DOM is a small application programming interface (API), which provides a set of functions that are common across Adobe Creative Suite 2 applications. These include functions to open files, execute scripts, and print files. For details of the function set, see the Cross-DOM API Reference.

You can access Cross-DOM functions in any Creative Suite 2 script by prefixing the function name with the *namespace specifier* for the target application (see Namespace specifiers). For example, a Photoshop CS2 script can call `indesign.open(file)` to open a file in InDesign CS2, or `golive.open(file)` to open a file in GoLive CS2.

The Cross-DOM functions for each application are implemented in JavaScript. You can see the implementation for each installed application by reading its associated startup script in the Adobe startup folder. For example, Illustrator CS2 defines `illustrator.open()` in the `illustrator-n.jsx` startup script (where *n* is the version number of the installed application).

- In Windows, the application start-up scripts are found in:

  `%CommonProgramFiles%\Adobe\StartupScripts`

- In Mac OS, the application start-up scripts are found in:

  `/Library/Application Support/Adobe/StartupScripts/`

Note: This is *not* the location in which to store your own startup scripts; see Script Locations and Checking Application Installation.

Cross-DOM API Reference

All exported functions, including those of the Cross-DOM API, are invoked through the exporting application, identified by its *namespace specifier* (see Namespace specifiers). For example:

```
//execute an Illustrator script in version 12
illustrator12.executeScript(myAIScript);
```

A specifier with no version information invokes the highest installed version of the application. For example:

```
//execute a Photoshop script in the highest available version
photoshop.executeScript (myPSScript);
```

All Adobe Creative Suite 2 applications implement the following Cross-DOM functions:

Function	Description
`executeScript` `appspec.executeScript(script)`	Performs a JavaScript `eval` on the specified script. The entire document object model (DOM) of the target application is available to the script. Returns `undefined`.
`script`	A string containing the script to be evaluated.
`open` `appspec.open(files)`	Performs the equivalent of the target application's **File > Open** command on the specified files. Returns `undefined`.
`files`	A File Object or array of `File` objects. For applications that use compound documents, this should be a project file.

openAsNew *appspec*.openAsNew([*options*])	Performs the equivalent of the target application's **File > New** command. Returns `true` on success.

options	Optional. Application-specific creation options:
	Bridge: none
	Photoshop: none
	InDesign: creation options are:
	`(Boolean:showingWindow, ObjectOrString:documentPresets)`
	See the arguments for `documents.add()` in the *InDesign CS2 Scripting Reference*.
	Illustrator: creation options are:
	`([DocumentColorSpace:colorspace][, Number:width, Number:height])`
	See the arguments for `documents.add()` in the *Illustrator CS2 JavaScript Reference*.
	GoLive: creation options are:
	`([ObjectOrString:templateFile])`
	See the arguments for `app.newDocument()` in the *GoLive CS2 SDK JavaScript Reference*.

print *appspec*.print(*files*)	Performs the equivalent of the target application's **File > Print** command on the specified files. Returns `undefined`.
files	A File Object or array of `File` objects. For applications that use compound documents, this should be a project file.
quit *appspec*.quit()	Performs the equivalent of the target application's **File > Exit** or **File > Close** command. Returns `undefined`.
reveal *appspec*.reveal(*file*)	Gives the target application the operating-system focus, and, if the specified file is open in that application, brings it to the foreground. Returns `undefined`.
file	A File Object or string specifying a file that can be opened in the target application.

Application-Specific Exported Functions

Each Adobe Creative Suite 2 can provide application-specific functionality to all Creative Suite 2 scripts through a simple syntax. You can access exported functions in any Creative Suite 2 script by prefixing the function name with the namespace specifier for the target application (see Namespace specifiers). For example, Photoshop CS2 exports the `photomerge` function, so a GoLive CS2 script can directly call `photoshop.photomerge(files)`.

The only difference between Cross-DOM functions and the application-specific exported functions is that all applications expose the same set of Cross-DOM functions, whereas each application exposes its own set of application-specific functions. Each application determines the extent of its exported functionality. Some applications provide extensive support for exported functions, others less.

For details of additional functions that are exported by individual applications, refer to the startup scripts for those applications. The application startup scripts are named `appname-n.jsx`, where *n* is the version number of the installed application, and found at:

`%CommonProgramFiles%\Adobe\StartupScripts` (in Windows)

`/Library/Application Support/Adobe/StartupScripts/` (in Mac OS)

Note: This is not the location in which to store your own startup scripts; see Script Locations and Checking Application Installation.

Communicating Through Interapplication Messages

Bridge provides an application programming interface (API) that defines a communication protocol between Adobe Creative Suite 2 applications. This provides the most general mechanism for communication between applications. A messaging-enabled application can launch another messaging-enabled application, and send or receive scripts to effect certain actions. For example, from within Bridge, a script can launch Photoshop, and then send a script to Photoshop that requests a photomerge operation.

While the exported functions allow specific access to certain capabilities of the application, the script in an interapplication message allows full access to the target application's document object model (DOM), in addition to all cross-DOM and application exported functions.

The messaging API defines the BridgeTalk Class, whose static properties and functions provide access to environmental information relevant for communication between applications. You can instantiate this class to create a BridgeTalk Message Object, which encapsulates a message and allows you to send it to another application. For details of these objects, see Interapplication Message API Reference.

Sending messages

To send a script or other data to another application, you must create and configure a BridgeTalk Message Object. This object contains the data to be sent (generally a script to be executed in the target application), and also specifies how to handle the response.

This simple example walks through the steps of sending a script from Bridge to Photoshop CS2, and receiving a response.

Step 1: Check that the target application is installed

Before you can actually send a message, you must check that the required version of the target application is installed. The function getSpecifier, available in the global namespace through the BridgeTalk Class, provides this information.

For example, this code, which will send a message to Bridge as part of a script being executed by Photoshop CS2, checks that the required version of Bridge is installed:

```
var targetApp = BridgeTalk.getSpecifier( "bridge", "1");
if( targetApp ) {
   // construct and send message
}
```

When you send the message, the messaging framework automatically starts the target application, if it is not already running.

Step 2: Construct a message object

The next step is to construct a message to send to the application. You do this by creating a BridgeTalk Message Object, and assigning values to its properties. You must specify the target application and the message body, which is usually a script packaged into a string.

Scripts sent in messages can be very complex, and can use the full DOM of the target application. This example defines a message script that accesses the Bridge DOM to request the number of files or folders found in a specific folder:

```
var bt = new BridgeTalk; // create a new BridgeTalk message object
bt.target  = "bridge"; // send this message to the Bridge application
```

```
// the script to evaluate is contained in a string in the "body" property
bt.body = "app.browseTo(Folder('C/MyPhotos'));
             app.document.target.children.length;"
```

Step 3: Specify how to handle a response

If you want to handle a response for this message, or use the data that is returned from the script's evaluation, you must set up the response-handling mechanism before you send the message. You do this by defining the onResult callback in the message object.

The response to a message is, by default, the result of evaluation of the script contained in that message's `body` property. The target application might define some different kind of response; see Receiving messages.

When the target has finished processing this message, it looks for an `onResult` callback in the message object it received. If it is found, the target automatically invokes it, passing it the response. The response is packaged into a string, which is in turn packaged into the `body` property of a new message object. That message object is the argument to your `onResult` callback function.

This handler, for example, processes the returned result using a script-defined `processResult` function.

```
bt.onResult = function(returnBtObj)
   { processResult(returnBtObj.body); }
```

If you want to handle errors that might arise during script processing, you can define an onError callback in the message object. For more information, see Handling responses from the message target.

Step 4: Send the message

To send the message, call the message object's `send` method. You do not need to specify where to send the message to, since the target application is set in the message itself.

```
bt.send();
```

The complete script looks like this:

```
// script to be executed in Photoshop CS2
#target "photoshop"
// check that the target app is installed
var targetApp = BridgeTalk.getSpecifier( "bridge", "1");
if( targetApp ) {
   // construct a message object
   var bt = new BridgeTalk;
   // the message is intended for Bridge
   bt.target  = "bridge";
   // the script to evaluate is contained in a string in the "body" property
   bt.body = "app.browseTo('C/MyPhotos');";
             app.document.target.children.length;"
   // define result handler callback
   bt.onResult = function(returnBtObj) {
      processResult(returnBtObj.body); } //fn defined elsewhere
   // send the message
   bt.send();
}
```

Receiving messages

An application can be the target of a message; that is, it receives an unsolicited message from another application. An unsolicited message is handled by the static `BridgeTalk.`onReceived callback function in the target application. See Handling unsolicited messages.

An application that sends a message can receive *response messages*; that is, messages that come as the result of requesting a response when a message was sent. These can be:

- The result of an error in processing the message
- A notification of receipt of the message
- Intermediate responses
- The final result of processing the message.

All of these response messages are sent automatically by the target application, and are handled by callbacks defined in the sending message object. For details, see Handling responses from the message target.

Handling unsolicited messages

To specify how the application should handle unsolicited incoming messages, define a callback handler function in the static onReceived property of the `BridgeTalk` class. This function takes a single argument, a BridgeTalk Message Object.

The default behavior of the `onReceived` handler is to evaluate the `body` of the received message with JavaScript, and return the result of that evaluation. (The result of evaluating a script is the result of the last line of the script.) To return the result, it creates a new message object, encapsulates the result in a string in the `body` property of that object, and passes that object to the onResult callback defined in the original message.

If an error occurs on evaluation, the default `onReceived` handler returns the error information using a similar mechanism. it creates a new message object, encapsulates the error information in a string in the `body` property of that object, and passes that object to the onError callback defined in the original message.

To change the default behavior set the `BridgeTalk.onReceived` property to a function definition in the following form:

```
BridgeTalk.onReceived = function( bridgeTalkObject ) {
  // callback definition here
  };
```

- The `body` property of the received message object contains the received data.
- The function can return any type.

The function that you define does not need to explicitly create and return a `BridgeTalk` message object. The messaging framework creates a new `BridgeTalk` message object, and packages the return value of the `onReceived` handler as a string in the `body` property of that object.

Return values are flattened into a string using the Unicode Transformation Format-8 (UTF-8) encoding. If the function does not specify a return value, the resulting string is `"undefined"`.

The result object is transmitted back to the sender if the sender has implemented an `onResult` callback for the original message.

➤ Message handling examples

This example shows the default mechanism for handling unsolicited messages received from other applications. This simple handler executes the message's data as a script and returns the results of that execution.

```
BridgeTalk.onReceived = function (message) {
   return eval( message.body );
}
```

This example shows how you might extend the receive handler to process a new type of message.

```
BridgeTalk.onReceived = function (message) {
   switch (message.type) {
      case "Data":
         return processData( message );
         break;
      default:  //"ExtendScript"
         return eval( mesage.body );
      }
   }
```

Handling responses from the message target

To handle responses to a message you have sent, you define callback handler functions in the message object itself. The target application cannot send a response message back to the sender unless the message object it received has the appropriate callback defined.

When your message is received by its target, the target application's static `BridgeTalk` object's onReceived method processes that message, and can invoke one of the message object's callbacks to return a response. In each case, the messaging framework packages the response in a new message object, whose target application is the sender. Your callback functions receive this response message object as an argument.

A response message can be:

- The result of an error in processing the message. This is handled by the onError callback.

  If an error occurs in processing the message `body` (as the result of a JavaScript syntax error, for instance), the target application invokes the `onError` callback, passing a response message that contains the error code and error message. If you do not have an `onError` callback defined, the error is completely transparent. It can appear that the message has not been processed, since no result is ever returned to the `onResult` callback.

- A notification of receipt of the message. This is handled by the onReceived callback.

  Message sending is asynchronous. Getting a `true` result from the send method does not guarantee that your message was actually received by the target application. If you want to be notified of the receipt of your message, define the onReceived callback in the message object. The target sends back the original message object to this callback, first replacing the `body` value with an empty string.

- Intermediate responses. These are handled by the onResult callback.

  The script that you send can send back intermediate responses by invoking the original message object's sendResult method. It can send data of any type, but that data is packaged into a `body` string in a new message object, which is passed to your callback. See Passing values between applications.

- The final result of processing the message. This is handled by the onResult callback.

  When it finishes processing your message, the target application can send back a result of any type. If you have sent a script, and the target application is using the default `BridgeTalk.onReceived` callback to process messages, the return value is the final result of evaluating that script. In any case, the return value is packaged into a `body` string in a new message object, which is passed to your callback. See Passing values between applications.

The following examples demonstrate how to handle simple responses and multiple responses, and how to integrate error handling with response handling.

➤ Example: Receiving a simple response

In this example, an application script asks Bridge to find out how many files and folders are in a certain folder, which the evaluation of the script returns. (The default `BridgeTalk.onReceived` method processes this correctly.)

The `onResult` method saves that number in `fileCountResult`, a script-defined property of the message, for later use.

```
var bt = new BridgeTalk;
bt.target = "bridge";
bt.body = "app.browseTo('C/MyPhotos');
            app.document.target.children.length;"
bt.onResult = function( retObj ) {
  processFileCount(retObj.body);
}
bt.send();
```

➤ Example: Handling any error

In this example, the `onError` handler re-throws the error message within the sending application.

```
var bt = new BridgeTalk;
bt.onError = function (btObj) {
  var errorCode = parseInt (btObj.headers ["Error-Code"]);
  throw new Error (errorCode, btObj.body);
}
```

➤ Example: Handling expected errors and responses

This example creates a message that asks Bridge to return XMP metadata for a specific file. The `onResult` method processes the data using a script-defined `processFileSize` function. Any errors are handled by the `onError` method. For example, if the file requested is not an existing file, the resulting error is returned to the `onError` method.

```
var bt = new BridgeTalk;
bt.target = "bridge";
bt.body = "var tn = new Thumbnail('C/MyPhotos/temp.jpg');
            tn.metadata.FileSize;"
bt.onResult = function( resultMsg ) {
  processFileSize(resultMsg.body);
}
bt.onError = function( errorMsg ) {
  var errCode = parseInt (errorMsg.headers ["Error-Code"]);
  throw new Error (errCode, errorMsg.body);
}
bt.send();
```

➤ Setting up a target to send multiple responses

This example integrates the sending of multiple responses with the evaluation of a message body. It sets up a handler for message such as the one sent in the following example.

The target application (Bridge) defines a static onReceived method to allow for a new type of message, which it calls an *iterator*. An `iterator` type of message expects the `message.body` to use the iteration variable `i` within the script, so that different results are produced for each pass through the `while` loop. Each result is sent back to the sending application with the sendResult method. When the `message.body` has finished processing its task, it sets a flag to end the `while` loop.

```
// Code for processing the message and sending intermediate responses
// in the target application (Bridge)
   BridgeTalk.onReceived = function (message){
      switch (message.type) {
         case "iterator":
            done = false;
            i = 0;
            while (!done) {
               // the message.body uses "i" to produce different results
               // for each execution of the message.
               // when done, the message.body sets "done" to true
               // so this onReceived method breaks out of the loop.
                message.sendResult(eval(message.body));
               i++; }
            break;
         default:  //"ExtendScript"
            return eval( message.body );
      }
   }
```

➤ Example: Setting up a sender to receive multiple responses

This example sends a message of the type `iterator`, to be handled by the onReceived handler in the previous example.

The sending application creates a message whose script (contained in the `body` string) iterates through all files in a specific folder (represented by a Bridge `Thumbnail` object), using the iterator variable `i`. For each file in the folder, it returns `FileSize` metadata. For each contained folder, it returns -1. The last executed line in the script is the final result value for the message.

The onResult method of the message object receives each intermediate result, stores it into an array, `resArr`, and processes it immediately using a script-defined function `processInterResult`.

```
// Code for send message and handling response
// in the sending application (any CS2 application)
   var idx = 0;
   var resArr = new Array;

   bt = new BridgeTalk;
   bt.target = "bridge";
   bt.type = "iterator";

   bt.body = "
      var fld = new Thumbnail(Folder('C/Junk'));
      if (i == (fld.children.length - 1))
         done = true; //no more files, end loop
```

```
    tn = fld.children[i];
    if (tn.spec.constructor.name == 'File')
       md = tn.metadata.FileSize;
    else md = -1;
    ";
  // store intermediate results
  bt.onResult = function(rObj) {
    resArr[idx] = rObj.body;
    processInterResult(resArr[idx]);
    idx++;};

  bt.onError = function(eObj) {
    bt.error = eObj.body };

  bt.send();
```

Passing values between applications

The BridgeTalk.onReceived static callback function can return values of any type. The messaging framework, however, packages the response into a response message, and passes any returned values in the message body, first converting the result to a UTF-8-encoded string.

Passing simple types

When your message object's onResult callback receives a response, it must interpret the string it finds in the body of the response message to obtain a result of the correct type. Results of various types can be identified and processed as follows:

Number	JavaScript allows you to access a string that contains a number directly as a number, without doing any type conversion. However, be careful when using the plus operator (+), which works with either strings or numbers. If one of the operands is a string, both operands are converted to strings and concatenated.
String	No conversion is required.
Boolean	The result string is either "true" or "false". You can convert it to a true boolean by evaluating it with the eval method.
Date	The result string contains the date in the form: "*dow mmm dd yyyy hh*:*mm*:*ss* GMT-*nnnn*". For example "Wed Jun 23 2004 00:00:00 GMT-0700".
Array	The result string contains a comma delimited list of the elements of the array. For example, If the result array is [12, "test", 432], the message framework flattens this into the string "12,test,432". As an alternative to simply returning the array, the message target can use the toSource method to return the code used to create the array. In this case, the sender must reconstitute the array by using the eval method on the result string in the response body. See discussion below.

Passing complex types

When returning complex types (arrays and objects), the script that you send construct a result string itself, using the `toSource` method to return the code used to create the array or object. In this case, the sender must reconstitute the array or object by using the `eval` method on the result string in the response body.

➤ Passing an array with toSource and eval

For example, the following code sends a script that returns an array in this way. The `onResult` callback that receives the response uses eval to reconstruct the array.

```
var bt = new BridgeTalk;
bt.target = "bridge";
// the script passed to the target application
// needs to return the array using "toSource"
bt.body = "var arr = [10, "this string", 324];
            arr.toSource();"
bt.onResult = function(resObj) {
   // use eval to reconstruct the array
   arr = eval(resObj.body);
   // Now you can access the returned array
   for (i=0; i< arr.length(); i++)
   doSomething(arr[i]);
}
//launch the Bridge if it's not already running
if (!BridgeTalk.isRunning("bridge"))
   BridgeTalk.launch("bridge");
// send the message
bt.send();
```

➤ Passing an object with toSource and eval

This technique is the only way to pass objects between applications. For example, this code sends a script that returns an object containing some of the metadata for a specific file, and defines an `onResult` callback that receives the object.

```
var bt = new BridgeTalk;
bt.target = "bridge";

//the script passed to the target application
// returns the object using "toSource"
bt.body = "var tn = new Thumbnail(File('C:\\Myphotos\\photo1.jpg'));
            var md = {fname:tn.metadata.FileName,
               fsize:tn.metadata.FileSize};
            md.toSource();"
//For the result, use eval to reconstruct the object
bt.onResult = function(resObj) {
    md = bt.result = eval(resObj.body);
   // Now you can access fname and fsize properties
   doSomething (md.fname, md.fsize);
}
//launch the Bridge if it's not already running
if (!BridgeTalk.isRunning("bridge"))
   BridgeTalk.launch("bridge");
// send the message
bt.send();
```

➤ Passing a DOM object

You can send a script that returns a DOM object, but the resulting object contains only those properties that were accessed within the script. For example, the following script requests the return of the Bridge DOM `Thumbnail` object. Only the properties `path` and `spec` are accessed by the script, and only those properties are returned:

```
var bt = new BridgeTalk;
bt.target = "bridge";
//set up the script passed to the target application
// to return the array using "toSource"
bt.body = "var tn = new Thumbnail(File('C:\\Myphotos\\photo1.jpg'));
   var s = tn.spec; var p = tn.path;
   tn.toSource();"
//For the result, use eval to reconstruct the object
bt.onResult = function(resObj) {
   // use eval to reconstruct the object
   tn = eval(resObj.body);
   // Now the script can access tn.spec and tn.path,
   // but no other properties of the Bridge DOM Thumbnail object
   doSomething (tn.spec, tn.path);
}
// send the message
bt.send();
```

Interapplication Message API Reference

This application programming interface (API) defines a communication protocol between Adobe Creative Suite 2 applications. These objects are available to all ExtendScript scripts when any of the applications is loaded.

The messaging protocol is extensible. Although it is primarily designed to send scripts, you can use it to send other kinds of data.

The messaging API defines the `BridgeTalk` class. Static properties and methods of the class provide access to environmental information relevant for communication between applications. Instantiate the class to create a `BridgeTalk` message object, which encapsulates the message itself. For discussion and examples, see Communicating Through Interapplication Messages.

BridgeTalk Class

Static properties and methods of this class provide a way for your script to determine basic messaging system information before you create any specific message objects. Static methods allow you to check if an application is installed and is already running, and to launch the application. A callback defined on the class determines how the application processes incoming messages.

You can access static properties and methods in the `BridgeTalk` class, which is available in the global namespace. For example:

```
var thisApp = BridgeTalk.appName;
```

Note: You must instantiate `BridgeTalk` class to create the `BridgeTalk` message object, which is used to send message packets between applications. Dynamic properties and methods can be accessed only in instances.

BridgeTalk class properties

The `BridgeTalk` class provides these static properties, which are available in the global namespace:

`appLocale`	String	The locale of this application, the *locale* portion of an application specifier; see Application specifiers. When a message is sent, this is the locale of the sending application. Read/Write.
`appName`	String	The name of this application, the *appname* portion of an application specifier; see Application specifiers. When a message is sent, this is the name of the sending application. Read/Write.
`appVersion`	String	The version number of this application, the *version* portion of an application specifier; see Application specifiers. When a message is sent, this is the version of the sending application. Read/Write.

onReceived	Function	A callback function that this application applies to unsolicited incoming messages. The default function evaluates the body of the received message and returns the result of evaluation. To change the default behavior, set this to a function definition in the following form:
		`BridgeTalk.onReceived = function( bridgeTalkObject ) {`
		`   // act on received message`
		`};`
		The `body` property of the received message object contains the received data. The function can return any type. See Handling unsolicited messages.
		Note: This function is *not* applied to a message that is received in response to a message sent from this application. Response messages are processed by the `onResult`, `onReceived`, or `onError` callbacks associated with the sent message.

BridgeTalk class functions

The `BridgeTalk` class provides these static properties, which are available in the global namespace:

bringToFront `BridgeTalk.bringToFront (`*app*`)`	Brings all windows of the specified application to the front of the screen. In Mac OS, an application can be running but have no windows open. In this case, calling this function might or might not open a new window, depending on the application. For Bridge, it opens a new browser window.
app	A specifier for the target application; see Application specifiers.
getSpecifier `BridgeTalk.getSpecifier (app,[`*version*`],[`*locale*`])`	Returns a complete specifier (see Application specifiers) for a messaging-enabled application version installed on this computer, or `null` if the requested version of the application is not installed.
app	The base name of the application to search for.

version	Optional. The specific version number to search for. If `0` or not supplied, returns the most recent version. If negative, returns the highest version up to and including the absolute value.
	If a major version is specified, returns the highest minor-version variation. For example, if Photoshop CS2 9, 9.1, and 10 are installed:
	`BridgeTalk.Specifier( "photoshop", "9" )`
	`   => ["photoshop-9.1"]`

locale	Optional. The specific locale to search for. If not supplied and multiple language versions are installed, returns the version for the current locale.

➤ Examples

Assuming installed applications include Photoshop CS2 8.0 `en_US`, Photoshop CS2 8.5 `de_DE`, Photoshop CS2 9.0 `de_DE`, and Photoshop CS2 9.5 `de_DE`, and that the current locale is `en_US`:

```
BridgeTalk.getSpecifier ("photoshop");
   => ["photoshop-8.0-en_US"]
BridgeTalk.getSpecifier ("photoshop", 0, "en_US");
   => ["photoshop-8.0-en_US"]
BridgeTalk.getSpecifier ("photoshop", 0, "de_DE");
   => ["photoshop-9.5-de_DE"]
BridgeTalk.getSpecifier ("photoshop", -9.2, "de_DE");
   => ["photoshop-9.0-de_DE"]
BridgeTalk.getSpecifier ("photoshop", 8);
   => ["photoshop-8.0-us_EN"]
BridgeTalk.getSpecifier ("photoshop", 8, "de_DE");
   => ["photoshop-8.5-de_DE"]
```

`getTargets`

`BridgeTalk.getTargets ([version],[locale])`

Returns an array of Application specifiers for messaging-enabled applications installed on this computer.

- If `version` is supplied, returns the base name plus the version information.
- If `locale` is supplied, returns the full names, with both version and locale information.
- If neither `version` nor `locale` is supplied, returns base specifiers with neither version nor locale information.

`version` — Optional. The specific version number to search for, or `null` to return the most recent version, with version information.

- Specify only a major version number to return the highest minor-version variation. For example, if Photoshop CS2 9, 9.1, and 10 are installed:

```
BridgeTalk.getTargets( "9" )
    => [photoshop-9.1]
```

- Specify a negative value to return all versions up to the absolute value of the version number. For example:

```
BridgeTalk.getTargets( "-9.1" )
    => [photoshop-9.0,photoshop-9.1]
```

`locale` — Optional. The specific locale to search for, or `null` to return applications for all locales, with locale information.

If not supplied when *`version`* is supplied, returns specifiers with version information only.

➤ *Examples*

Assuming installed applications include Photoshop CS2 8.0 `en_US`, Photoshop CS2 9.0 `de_DE`, and Illustrator CS2 9.0 `de_DE`:

```
BridgeTalk.getTargets();
  => [photoshop,illustrator]
BridgeTalk.getTargets( "8.0" )
  => [photoshop-8.0]
BridgeTalk.getTargets( null )
  => [photoshop-9.0, illustrator-9.0]
BridgeTalk.getTargets( null, "en_US" )
  => [photoshop-8.0-en_US, illustrator-9.0-en_US]
BridgeTalk.getTargets( null, null )
  => [photoshop-8.0-en_US, photoshop-9.0-de_DE, illustrator-9.0-en_US]
BridgeTalk.getTargets( "9.0", null )
  => [photoshop-9.0-de_DE, illustrator-9.0-en_US]
```

`isRunning` `BridgeTalk.isRunning (`*`app`*`)`	Returns `true` if the given application is running and active on the local computer.
`app`	A specifier for the target application; see Application specifiers.
`launch` `BridgeTalk.launch (`*`app`* `[, `*`where`*`])`	Launches the given application on the local computer. Returns `undefined`. It is not necessary to launch an application explicitly in order to send it a message. Sending a message to an application that is not running automatically launches it.
`app`	A specifier for the target application; see Application specifiers.
`where`	Optional. If the value `"background"` is specified, the application’s main window is not brought to the front of the screen.
`pump` `BridgeTalk.pump ()`	Checks all active messaging interfaces for outgoing and incoming messages, and processes them if there are any. Returns `true` if any messages have been processed, `false` otherwise. (Most applications have a message processing loop that continually checks the message queues, so use of this method is rarely required.)

BridgeTalk Message Object

The message object defines the basic communication packet that is sent between applications. Its properties allow you to specify the receiving application (the `target`), the data to send to the target (the `body`), and the `type` of data that is sent. The messaging protocol is extensible; it allows you to define new types of data for the type property, and to send and receive arbitrary additional information with the `headers` property.

BridgeTalk message object constructor

Create a new message object using a simple constructor:

```
var bt = new BridgeTalk;
```

Before you send a message to another application, you must set the `target` property to the receiving application, and the `body` property to the data message (typically a script) you want to send.

BridgeTalk message object properties

body	String	The data payload of the message. Read/Write. • If this is an unsolicited message to another application, typically contains a script packaged as a string. The target application's full document object model (DOM) is available within the script. • If this message is a result returned from the static `BridgeTalk` onReceived method of a target application, directed to an onResult callback in this object, contains the return result from that method flattened into a string. See Passing values between applications. • If this message contains an error notification for the onError callback, contains the error message.

headers	Object	A JavaScript object containing script-defined headers. Read/Write.
		Use this property to define custom header data to send supplementary information between applications. You can add any number of new headers. The headers are name/value pairs, and can be accessed with the JavaScript dot notation (*msgObj*.headers.*propName*), or bracket notation (*msgObj*.headers[*propName*]). If the header name conforms to JavaScript symbol syntax, use the dot notation. If not, use the bracket notation.
		The pre-defined header `["Error-Code"]` is used to return error messages to a sender. See below for pre-defined error code.
		Examples of setting headers:
		`bt.headers.info = "Additional Information";`
		`bt.headers ["Error-Code"] = 8;`
		Examples of getting header values:
		`var info = bt.headers.info;`
		`var error = bt.headers ["Error-Code"];`

sender	String	The application specifier for the sending application (see Application specifiers). Read/Write.

target	String	The application specifier for the target, or receiving, application (see Application specifiers). Read/Write.
timeout	Number	The number of milliseconds before the message times out. Read/Write. If a message has not been removed from the input queue for processing before this time elapses, the message is discarded. If the sender has requested that errors be transferred back by defining an `onError` callback for the message, the target application sends a timeout message back to the sender.
type	String	The message type, which indicates what type of data the `body` contains. Read/Write. Default is `ExtendScript`. You can define a type for script-defined data. If you do so, the target application must have a static `BridgeTalk` onReceived method that checks for and processes that type.

BridgeTalk message object callbacks

onError	Function	A callback function that the target application invokes to return an error response to the sender. It can send JavaScript run-time errors or exceptions, or C++ exceptions.
		To define error-response behavior, set this to a function definition in the following form:
		`bridgeTalkObj.onError = function( errorMsgObject ) {`
		`   // error handler defined here`
		`};`
		The `body` property of the received message object contains the error message, and the `headers` property contains the error code in its `Error-Code` property. See Messaging Error Codes.
		The function returns `undefined`.

onReceived	Function	A callback function that the target application invokes to confirm that the message was received. (Note that this is different from the static onReceived method of the BridgeTalk class that handles unsolicited messages.)
		To define a response to receipt notification, set this to a function definition in the following form:
		`bridgeTalkObj.onReceived = function( origMsgObject ) {`
		`   // handler defined here`
		`};`
		The target passes back the original message object, with the body property set to the empty string.
		The function returns undefined.

onResult	Function	A callback function that the target application invokes to return a response to the sender. This can be an intermediate response or the final result of processing the message.
		To handle the response, set this to a function definition in the following form:
		`bridgeTalkObj.onResult = function( responseMsgObject ) {`
		`   // handler defined here`
		`};`
		The target passes a new message object, with the body property set to the result string. This is the result of the target application's static BridgeTalk onReceived method, packaged as a UTF-8-encoded string. See Passing values between applications.

BridgeTalk message object functions

send *bridgeTalkObj*.send ()	Sends this message to the target application. Returns true if the message could be sent immediately, false if it could not be sent or was queued for sending later. If the target application is not running and the message contains a body, the messaging system automatically launches the target application. In this case, the message is queued rather than sent immediately, and this method returns false. The message is processed once the application is running. Sending the message does not guarantee that the target actually receives it. You can request notification of receipt by defining an onReceived callback for this message object. (Note that this is different from the static onReceived method of the BridgeTalk class that handles unsolicited messages.)
sendResult *bridgeTalkObj*.sendResult (*result*)	When processing an unsolicited message, the static BridgeTalk onReceived method can return an intermediate result to the sender by calling this method in the received message object. It invokes the onResult callback of the original message, passing a new message object containing the specified *result* value. This allows you to send multiple responses to messages. Returns true if the received message has an onResult callback defined and the response message can be sent, false otherwise.
result	You can send data of any type as the result value. The messaging framework creates a BridgeTalk Message Object, and flattens this value into a string which it stores in the body of that message. See Passing values between applications.

Messaging Error Codes

The interapplication messaging protocol defines the following error codes, which are compatible with ExtendScript error codes. Negative values indicate unrecoverable errors that cause ExtendScript to terminate a running script.

1	General error
8	Syntax error
20	Bad argument list
27	Stack overrun
-28	Out of memory
-29	Uncaught exception
31	Bad URI
32	Cannot perform requested action
-33	Internal error
-36	Not yet implemented
41	Range error
44	Cannot convert
47	Type mismatch
48	File or folder does not exist
49	File of folder already exists
50	I/O device is not open
51	Read past EOF
52	I/O error
53	Permission denied
54	JavaScript execution
56	Cannot connect
57	Cannot resolve reference
58	I/O timeout
59	No response

Sample Workflow Automation Scripts

Sample workflow automation scripts are provided with Adobe Creative Suite 2, at the following locations:

- Windows: `C:\Program Files\Common Files\Adobe\StartupScripts`
  Mac OS: `/Library/Application Support/Adobe/StartupScripts`

  ```
  AdobeLibrary1.jsx
  AdobeLibrary2.jsx
  AdobeScriptManager.jsx
  ```

- Windows: `C:\Program Files\Common Files\Adobe\StartupScripts\Workflow Automation Scripts`
  Mac OS: `/Library/Application Support/Adobe/StartupScripts/Workflow Automation Scripts`

  ```
  ContactSheet_ID.jsx
  ExportToFlash_AI.jsx
  ImportCameraImages_BR.jsx
  ```

These samples demonstrate various kinds of scripting usage and application interactions. The code can be used as a model of scripting style, and can be modified and expanded for your own use.

Script Name	**Description**	**Photoshop**	**Illustrator**	**InDesign**	**GoLive**	**Bridge**
Contact sheet (InDesign)	Generates a contact sheet from selected InDesign files, showing each page of each file.			X		X
Export to Flash	Exports a selected file to Flash (SWF) format.		X			X
Import files from camera	Automates import of files from a digital camera.					X
Script loader/manager	Utility that helps to manage existing or developer-created scripts and place them in the correct location in the application.	X	X	X	X	X

In addition to these, a sample GoLive extension, `BridgeSample`, Is included with the GoLive SDK in the GoLive installation directory, under `Adobe GoLive SDK 8.0/Samples`. This module demonstrates how to invoke Bridge from Adobe GoLive and communicate between Bridge and GoLive with messages.

10 ExtendScript Tools and Features

ExtendScript is the Adobe extended implementation of JavaScript, and is used by all Adobe applications that provide a scripting interface. In addition to implementing the JavaScript language according to the W3C specification, ExtendScript provides certain additional features and utilities.

- For help in developing, debugging, and testing scripts, ExtendScript provides:
  - The ExtendScript Toolkit, an interactive development and testing environment for ExtendScript.
  - A global debugging object, the Dollar ($) Object.
  - A reporting utility for ExtendScript elements, the ExtendScript Reflection Interface.
- In addition, ExtendScript provides these tools and features:
  - A localization utility for providing user-interface string values in different languages. See Localizing ExtendScript Strings.
  - Global functions for displaying short messages in dialog boxes. See User Notification Helper Functions.
  - An object type for specifying measurement values together with their units. See Specifying Measurement Values.
  - Tools for combining scripts, such as a `#include` directive, and `import` and `export` statements. See Modular Programming Support.
  - Support for extending or overriding math and logical operator behavior on a class-by-class basis. See Operator Overloading.
- ExtendScript provides a common scripting environment for all Adobe applications, and allows interapplication communication through scripts.
  - To identify specific Bridge-enabled applications, scripts must use Application and Namespace Specifiers.
  - Applications can run scripts automatically on startup. See Script Locations and Checking Application Installation.
  - For details about interapplication communication, see Interapplication Communication with Scripts.

The ExtendScript Toolkit

The ExtendScript Toolkit provides an interactive development and testing environment for ExtendScript in all Adobe applications that support scripting. It includes a full-featured, syntax-highlighting editor with Unicode capabilities and multiple undo/redo support. The Toolkit allows you to:

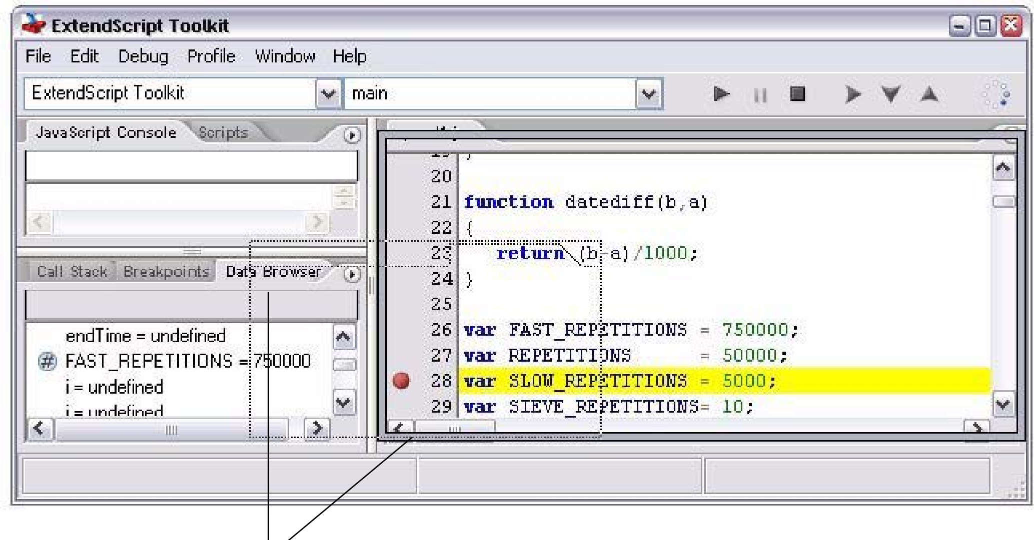
- Single-step through JavaScripts inside an Adobe application.
- Inspect all data for a running script.
- Set and execute breakpoints.

The Toolkit is the default editor for ExtendScript files, which use the extension `.jsx`. You can use the Toolkit to edit or debug scripts in JS or JSX files.

When you double-click a JSX file in the platform's windowing environment, the script runs in the Toolkit, unless it specifies a particular target application using the `#target` directive. For more information, see Selecting a debugging target and Preprocessor directives.

Configuring the Toolkit window

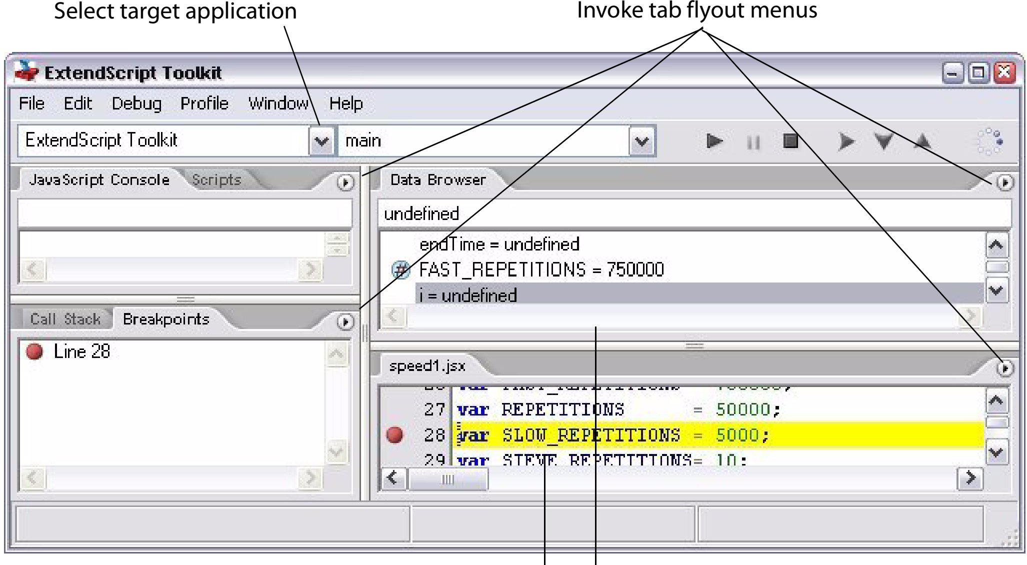
The ExtendScript Toolkit initially appears with a default arrangement of panes, containing a default configuration of tabs. You can adjust the relative sizes of the panes by dragging the separators up or down, or right or left. You can regroup the tabs. To move a tab, drag the label into another pane.



Drag a tab to a new pane Destination pane is highlighted, and the new tab is added to the tab stack

If you drag a tab so that the entire destination pane is highlighted, it becomes another stacked tab in that pane. If you drag a tab to the top or bottom of a pane (so that only the top or bottom bar of the destination pane is highlighted), that pane splits to show the tabs in a tiled format.

Select target application Invoke tab flyout menus



Split pane shows Browser and Editor tabs

Each tab has a flyout menu, accessed through the arrow icon in the upper right corner. The same menu is available as a context menu, which you invoke with a right click in the tab. This menu always includes a **Hide Pane** command to hide that pane. Use the **Window** menu to show a hidden pane, or to bring it to the front.

The Editor, which has a tab for each script, has an additional context menu for debugging, which appears when you right-click in the line numbers area.

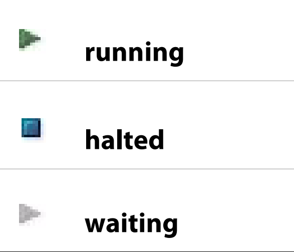
The Toolkit saves the current layout when you exit, and restores it at the next startup. It also saves and restores the open documents, the current positions within the documents, and any breakpoints that have been set.

- If you do not want to restore all settings on startup, hold SHIFT while the Toolkit loads to restore default settings. This reconnects to the last application and engine that was selected.
- If you want to restore the layout settings on startup, but not load the previously open documents, choose **Start with a clean workspace** in the Preferences dialog.

Selecting a debugging target

The Toolkit can debug multiple applications at one time. If you have more than one ExtendScript-enabled application installed, use the drop-down list at the upper left under the menu bar to select the target application. All installed applications that use ExtendScript are shown in this list. If you select an application that is not running, the Toolkit prompts for permission to run it.

All available engines in the selected target application are shown in a drop-down list to the right of the application list, with an icon that shows the current debugging status of that engine. A target application can have more than one ExtendScript engine, and more than one engine can be *active*, although only one is *current*. An active engine is one that is currently executing code, is halted at a breakpoint, or, having executed all scripts, is waiting to receive events. An icon by each engine name indicates whether it is *running*, *halted*, or *waiting* for input:



The current engine is the one whose data and state is displayed in the Toolkit's panes. If an application has only one engine, its engine becomes current when you select the application as the target. If there is more than one engine available in the target application, you can select an engine in the list to make it current.

When you open the Toolkit, it attempts to reconnect to the same target and engine that was set last time it closed. If that target application is not running, the Toolkit prompts for permission to launch it. If permission is refused, the Toolkit itself becomes the target application.

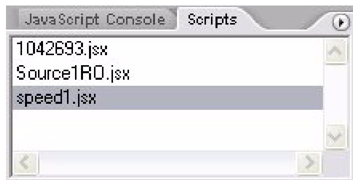
If the target application that you select is not running, the Toolkit prompts for permission and launches the application. Similarly, if you run a script that specifies a target application that is not running (using the `#target` directive), the Toolkit prompts for permission to launch it. If the application is running but not selected as the current target, the Toolkit prompts you to switch to it.

If you select an application that cannot be debugged in the Toolkit (such as Adobe Help), an error dialog reports that the Toolkit cannot connect to the selected application.

The ExtendScript Toolkit is the default editor for JSX files. If you double-click a JSX file in a file browser, the Toolkit looks for a `#target` directive in the file and launches that application to run the script; however, it first checks for syntax errors in the script. If any are found, the Toolkit displays the error in a message box and quits silently, rather than launching the target application. For example:

Selecting scripts

The Scripts tab offers a list of debuggable scripts for the target application, which can be JS or JSX files or (for some applications) HTML files that contain embedded scripts.



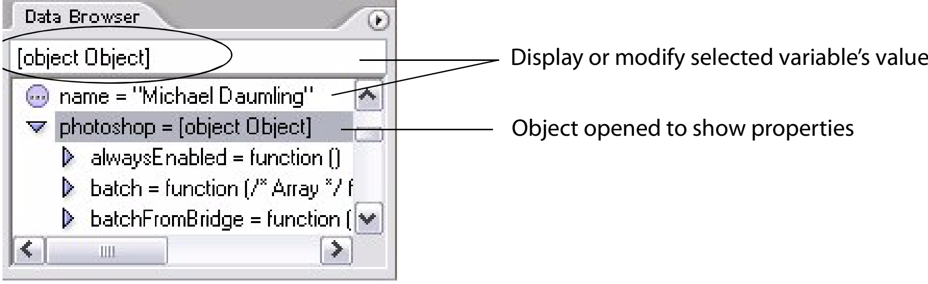
Select a script in this tab to load it and display its contents in the editor pane, where you can modify it, save it, or run it within the target application.

Tracking data

The Data Browser tab is your window into the JavaScript engine. It displays all live data defined in the current context, as a list of variables with their current values. If execution has stopped at a breakpoint, it shows variables that have been defined using `var` in the current function, and the function arguments. To show variables defined in the global or calling scope, use the Call Stack to change the context (see The call stack).

You can use the Data Browser to examine and set variable values.

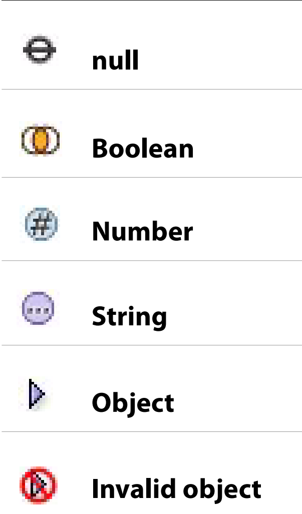
- Click a variable name to show its current value in the edit field at the top of the tab.
- To change the value, enter a new value and press ENTER. If a variable is read-only, the edit field is disabled.



The flyout menu for this tab lets you control the amount of data displayed:

- **Show Global Functions** toggles the display of all global function definitions.
- **Show Object Methods** toggles the display of all functions that are attached to objects. Most often, the interesting data in an object are its callable methods.
- **Show JavaScript Language Elements** toggles the display of all data that is part of the JavaScript language standard, such as the Array constructor or the Math object. An interesting property is the `__proto__` property, which reveals the JavaScript object prototype chain.

Each variable has a small icon that indicates the data type. An invalid object is a reference to an object that has been deleted. If a variable is undefined, it does not have an icon.

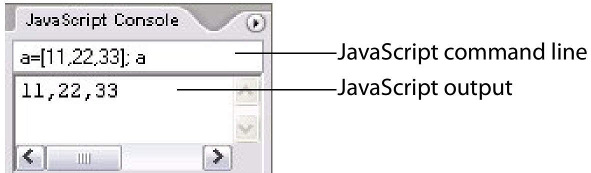


You can inspect the contents of an object by clicking its icon. The list expands to show the object's properties (and methods, if **Show Object Methods** is enabled), and the triangle points down to indicate that the object is open.

Note: In Photoshop CS2 the Data Browser pane is populated only during the debugging of a JavaScript program within Photoshop.

The JavaScript console

The JavaScript console is a command shell and output window for the currently selected JavaScript engine. It connects you to the global namespace of that engine.



The command line entry field accepts any JavaScript code, and you can use it to evaluate expressions or call functions. Enter any JavaScript statement on the command line and execute it by pressing ENTER. The statement executes within the stack scope of the line highlighted in the Call Stack tab, and the result appears in the output field.

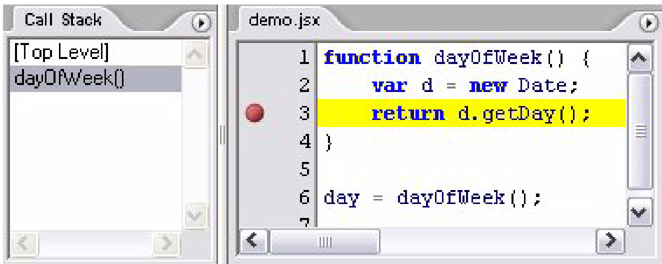
- The command line input field keeps a command history of 32 lines. Use the Up and Down Arrow keys to scroll through the previous entries.
- Commands entered in this field execute with a timeout of one second. If a command takes longer than one second to execute, the Toolkit generates a timeout error and terminates the attempt.

The output field is standard output for JavaScript execution. If any script generates a syntax error, the error is displayed here along with the file name and the line number. The Toolkit displays errors here during its own startup phase. The tab's flyout menu allows you to clear the contents of the output field and change the size of the font used for output.

The call stack

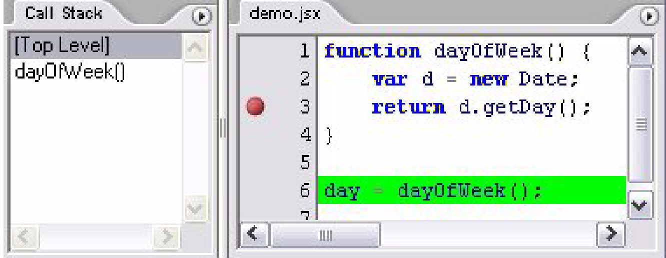
The Call Stack tab is active while debugging a program. When an executing program stops because of a breakpoint or runtime error, the tab displays the sequence of function calls that led to the current execution point. The Call Stack tab shows the names of the active functions, along with the actual arguments passed in to that function.

For example, this stack pane shows a break occurring at a breakpoint in a function `dayOfWeek`:



The function containing the breakpoint is highlighted in both the Call Stack and the Editor tabs.

You can click any function in the call hierarchy to inspect it. In the Editor, the line containing the function call that led to that point of execution is marked with a green background. In the example, when you select the line `[Top Level]` in the call stack, the Editor highlights the line where the `dayOfWeek` function was called.



Switching between the functions in the call hierarchy allows you to trace how the current function was called. The Console and Data Browser tabs coordinate with the Call Stack pane. When you select a function in the Call Stack:

- The Console pane switches its scope to the execution context of that function, so you can inspect and modify its local variables. These would otherwise be inaccessible to the running JavaScript program from within a called function.
- The Data Browser pane displays all data defined in the selected context.

The Script Editor

You can open any number of Script Editor tabs; each displays one Unicode source code document. The editor supports JavaScript syntax highlighting, JavaScript syntax checking, multiple undo and redo operations, and advanced search and replace functionality.

You can use the mouse or special keyboard shortcuts to move the insertion point or to select text in the editor.

Mouse navigation and selection

Click the left mouse button in the editor to move the position caret.

To select text with the mouse, click in unselected text, then drag over the text to be selected. If you drag above or below the currently displayed text, the text scrolls, continuing to select while scrolling. You can also double-click to select a word, or triple-click to select a line.

To initiate a drag-and-drop of selected text, click in the block of selected text, then drag to the destination. You can drag text from one editor pane to another. You can also drag text out of the Toolkit into another application that accepts dragged text, and drag text from another application into a Toolkit editor.

You can drop files from the Explorer or the Finder onto the Toolkit to open them in an editor.

Keyboard navigation and selection

Besides the usual keyboard input, the editor accepts these special movement keys. You can also select text by using a movement key while pressing SHIFT.

Enter	Insert a Line Feed character
Backspace	Delete character to the left
Delete	Delete character to the right
Left arrow	Move insertion point left one character
Right arrow	Move insertion point right one character
Up arrow	Move insertion point up one line; stay in column if possible
Down arrow	Move insertion point down one line; stay in column if possible
Page up	Move insertion point one page up
Page down	Move insertion point one page down
CTRL + Up arrow	Scroll up one line without moving the insertion point
Ctrl + Down arrow	Scroll down one line without moving the insertion point
CTRL + Page up	Scroll one page up without moving the insertion point
CTRL + page down	Scroll one page down without moving the insertion point
CTRL + Left arrow	Move insertion point one word to the left
CTRL + right arrow	Move insertion point one word to the right

Home	Move insertion point to start of line
End	Move insertion point to end of line
CTRL + Home	Move insertion point to start of text
CTRL + End	Move insertion point to end of text

The editor supports extended keyboard input via IME (Windows) or TMS (Mac OS). This is especially important for Far Eastern characters.

Syntax checking

Before running the new script or saving the text as a script file, you can check whether the text contains JavaScript syntax errors. Choose **Check Syntax** from the **Edit** menu or from the Editor's right-click context menu.

- If the script is syntactically correct, the status line shows "No syntax errors".
- If the Toolkit finds a syntax error, such as a missing quote, it highlights the affected text, plays a sound, and shows the error message in the status line so you can fix the error.

Debugging in the Toolkit

You can debug the code in the currently active Editor tab. Select one of the debugging commands to either run or to single-step through the program.

When you run code from the Editor, it runs in the current target application's selected JavaScript engine. The Toolkit itself runs an independent JavaScript engine, so you can quickly edit and run a script without connecting to a target application.

Evaluation in help tips

If you let your mouse pointer rest over a variable or function in an Editor tab, the result of evaluating that variable or function is displayed as a help tip. When you are not debugging the program, this is helpful only if the variables and functions are already known to the JavaScript engine. During debugging, however, this is an extremely useful way to display the current value of a variable, along with its current data type.

You can turn off the display of help tips using the **Display JavaScript variables** and **Enable UI help tips** checkboxes on the Help Options page of the Preferences dialog.

Controlling code execution

The debugging commands are available from the **Debug** menu, from the Editor's right-click context menu, through keyboard shortcuts, and from the toolbar buttons. Use these menu commands and buttons to control the execution of code when the JavaScript Debugger is active.

	Run **Continue**	**F5** (Windows) **Ctrl R** (Mac OS)	Starts or resumes execution of a script. Disabled when script is executing.
	Break	**Ctrl F5** (Windows) **Cmd .** (Mac OS)	Halts the currently executing script temporarily and reactivates the JavaScript Debugger. Enabled when a script is executing.
	Stop	**Shift F5** (Windows) **Ctrl K** (Mac OS)	Stops execution of the script and generates a runtime error. Enabled when a script is executing.
	Step Over	**F10** (Windows) **Ctrl S** (Mac OS)	Halts after executing a single JavaScript line in the script. If the statement calls a JavaScript function, executes the function in its entirety before stopping (do not step into the function).
	Step Into	**F11** (Windows) **Ctrl T** (Mac OS)	Halts after executing a single JavaScript line statement in the script or after executing a single statement in any JavaScript function that the script calls.
	Step Out	**Shift F11** (Windows) **Ctrl U** (Mac OS)	When paused within the body of a JavaScript function, resumes script execution until the function returns. When paused outside the body of a function, resumes script execution until the script terminates.

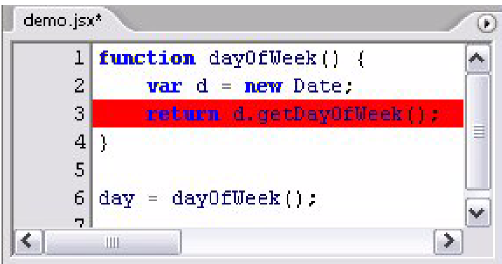
Visual indication of execution states

While the engine is running, an icon in the upper right corner of the Toolkit window indicates that the script is active.

When the execution of a script halts because the script reached a breakpoint, or when the script reaches the next line when stepping line by line, the Editor displays the current script with the current line highlighted in yellow.

```
demo.jsx
 1 function dayOfWeek() {
 2     var d = new Date;
 3     var day;
 4     try { // test dayOfWeek
 5         day = d.getDayOfWeek();
 6     }
 7     catch (error) { // no dayOfWeek
 8         day = -1;
 9     }
10     return day;
11 }
```

If the script encounters a runtime error, the Toolkit halts the execution of the script, displays the current script with the current line highlighted in red, displays the error message in the status line, and plays a sound.



Scripts often use a `try/catch` clause to execute code that may cause a runtime error, in order to catch the error programmatically rather than have the script terminate. You can choose to allow regular processing of such errors using the `catch` clause, rather than breaking into the debugger. To set this behavior, choose **Debug** > **Don't Break On Guarded Exceptions**. Some runtime errors, such as `Out Of Memory`, always cause the termination of the script, regardless of this setting.

Setting breakpoints

When debugging a script, it is often helpful to make it stop at certain lines so that you can inspect the state of the environment, whether function calls are nested properly, or whether all variables contain the expected data.

- To stop execution of a script at a given line, click to the left of the line number to set a breakpoint. A filled dot indicates the breakpoint.
- Click a second time to temporarily disable the breakpoint; the icon changes to an outline.
- Click a third time to delete the breakpoint. The icon is removed.

Some breakpoints need to be conditional. For example, if you set a breakpoint in a loop that is executed several thousand times, you would not want to have the program stop each time through the loop, but only on each 1000th iteration.

You can attach a condition to a breakpoint, in the form of a JavaScript expression. Every time execution reaches the breakpoint, it runs the JavaScript expression. If the expression evaluates to a nonzero number or `true`, execution stops.

To set a conditional breakpoint in a loop, for example, the conditional expression could be `"i >= 1000"`, which means that the program execution halts if the value of the iteration variable `i` is equal to or greater than 1000.

You can set breakpoints on lines that do not contain any code, such as comment lines. When the Toolkit runs the program, it automatically moves such a breakpoint down to the next line that actually contains code.

Breakpoint icons

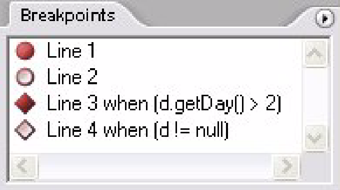
Each breakpoint is indicated by an icon to the left of the line number. The icon for a conditional breakpoint is a diamond, while the icon for an unconditional breakpoint is round. Disabled breakpoints are indicated by an outline icon, while active ones are filled.

	Unconditional breakpoint. Execution stops here.
	Unconditional breakpoint, disabled. Execution does not stop.

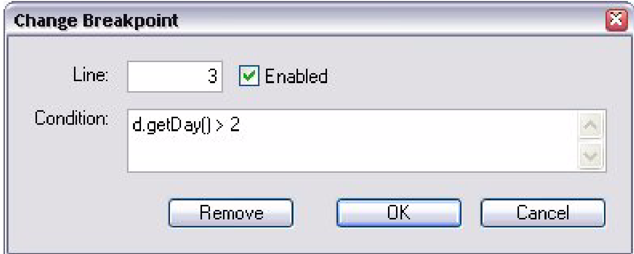
	Conditional breakpoint. Execution stops if the attached JavaScript expression evaluates to `true`.
	Conditional breakpoint, disabled. Execution does not stop.

The Breakpoints tab

The Breakpoints tab displays all breakpoints set in the current Editor tab. You can use the tab's flyout menu to add, change, or remove a breakpoint.



You can edit a breakpoint by double-clicking it, or by selecting it and choosing **Add** or **Change** from the context menu. A dialog allows you to change the line number, the breakpoint's enabled state, and the condition statement.



Whenever execution reaches this breakpoint, the debugger evaluates this condition. If it does not evaluate to `true`, the breakpoint is ignored and execution continues. This allows you to break only when certain conditions are met, such as a variable having a particular value.

Profiling

The Profiling tool helps you to optimize program execution. When you turn profiling on, the JavaScript engine collects information about a program while it is running. It counts how often the program executed a line or function, or how long it took to execute a line or function. You can choose exactly which profiling data to display.

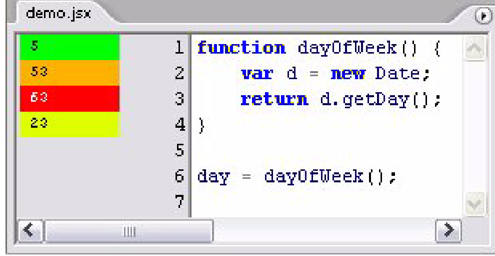
Because profiling significantly slows execution time, the **Profile** menu offers these profiling options.

Off	Profiling turned off. This is the default.
Functions	The profiler counts each function call. At the end of execution, displays the total to the left of the line number where the function header is defined.
Lines	The profiler counts each time each line is executed. At the end of execution, displays the total to the left of the line number. Consumes more execution time, but delivers more detailed information.

Add Timing Info	Instead of counting the functions or lines, records the time taken to execute each function or line. At the end of execution, displays the total number of microseconds spent in the function or line, to the left of the line number. This is the most time-consuming form of profiling.
No Profiler Data	When selected, do not display profiler data.
Show Hit Count	When selected, display hit counts.
Show Timing	When selected, display timing data.
Erase Profiler Data	Clear all profiling data.
Save Data As	Save profiling data as comma-separated values in a CSV file that can be loaded into a spreadsheet program such as Excel.

When execution halts (at termination, at a breakpoint, or due to a runtime error), the Toolkit displays this information in the Editor, line by line. The profiling data is color coded:

- Green indicates the lowest number of hits, or the fastest execution time.
- Red indicates trouble spots, such as a line that has been executed many times, or which line took the most time to execute.



This example displays timing information for the program, where the fastest line took 4 microseconds to execute, and the slowest line took 29 microseconds. The timing might not be accurate down to the microsecond; it depends on the resolution and accuracy of the hardware timers built into your computer.

Dollar ($) Object

This global ExtendScript object provides a number of debugging facilities and informational methods. The properties of the `$` object allow you to get global information such as the most recent run-time error, and set flags that control debugging and localization behavior. The methods allow you to output text to the JavaScript Console during script execution, control execution and other ExtendScript behavior programmatically, and gather statistics on object use.

Dollar ($) object properties

build	Number	The ExtendScript build number. Read only.
buildDate	Date	The date ExtendScript was built. Read only.
error	Error String	The most recent run-time error information, contained in a JavaScript `Error` object. Assigning error text to this property generates a run-time error; however, the preferred way to generate a run-time error is to throw an `Error` object.
flags	Number	Gets or sets low-level debug output flags. A logical AND of the following bit flag values: `0x0002` (2): Displays each line with its line number as it is executed. `0x0040` (64): Enables excessive garbage collection. Usually, garbage collection starts when the number of objects has increased by a certain amount since the last garbage collection. This flag causes ExtendScript to garbage collect after almost every statement. This impairs performance severely, but is useful when you suspect that an object gets released too soon. `0x0080` (128): Displays all calls with their arguments and the return value. `0x0100` (256): Enables extended error handling (see strict). `0x0200` (512): Enables the localization feature of the `toString` method. Equivalent to the localize property.
global	Object	Provides access to the global object, which contains the JavaScript global namespace.
level	Number	Enables or disables the JavaScript debugger. One of: 0: No debugging 1: Break on runtime errors 2: Full debug mode
locale	String	Gets or sets the current locale. The string contains five characters in the form *LL_RR*, where *LL* is an ISO 639 language specifier, and *RR* is an ISO 3166 region specifier. Initially, this is the value that the application or the platform returns for the current user. You can set it to temporarily change the locale for testing. To return to the application or platform setting, set to `undefined`, `null`, or the empty string.

localize	Boolean	Enable or disable the extended localization features of the built-in `toString` method. See Localizing ExtendScript Strings.
memCache	Number	Gets or sets the ExtendScript memory cache size in bytes.
objects	Number	The total count of all JavaScript objects defined so far. Read only.
os	String	The current operating system version. Read only.
screens	Array	An array of objects containing information about the display screens attached to your computer. Each object has the properties `left`, `top`, `right`, and `bottom`, which contain the four corners of each screen in global coordinates. A property `primary` is `true` if that object describes the primary display.
strict	Boolean	When `true`, any attempt to write to a read-only property causes a runtime error. Some objects do not permit the creation of new properties when `true`.
version	String	The version number of the ExtendScript engine as a three-part number and description; for example: "3.6.5 (debug)" Read only.

Dollar ($) object functions

about `$.about ()`	Displays the About box for the ExtendScript component, and returns the text of the About box as a string.
bp `$.bp ([`*condition*`])`	Executes a breakpoint at the current position. Returns `undefined`. If no condition is needed, it is recommended that you use the JavaScript `debugger` statement in the script, rather than this method.
condition	Optional. A string containing a JavaScript statement to be used as a condition. If the statement evaluates to `true` or nonzero when this point is reached, execution stops.
clearbp `$.clearbp ([`*line*`])`	Removes a breakpoint from the current script. Returns `undefined`.
line	Optional. The line at which to clear the breakpoint. If 0 or not supplied, clears the breakpoint at the current line number.
gc `$.gc ()`	Initiates garbage collection. Returns `undefined`.
getenv `$.getenv (`*envname*`)`	Returns the value of the specified environment variable, or `null` if no such variable is defined.
envname	The name of the environment variable.
list `$.list ([`*classname*`])`	Collects object information into a table and returns this table as a string. See Object statistics below.
classname	Optional. The type of object about which to collect information. If not supplied, collects information about all objects currently defined.

setbp `$.setbp ([line, condition])`	Sets a breakpoint in the current script. Returns `undefined`. If no arguments are needed, it is recommended that you use the JavaScript `debugger` statement in the script, rather than this method.
line	Optional. The line at which to stop execution. If 0 or not supplied, sets the breakpoint at the current line number.
condition	Optional. A string containing a JavaScript statement to be used for a conditional breakpoint. If the statement evaluates to `true` or nonzero when the line is reached, execution stops.
sleep `$.sleep (milliseconds)`	Suspends the calling thread for the given number of milliseconds. Returns `undefined`. During a sleep period, checks at 100 millisecond intervals to see whether the sleep should be terminated. This can happen if there is a break request, or if the script timeout has expired.
milliseconds	The number of milliseconds to wait.

summary
`$.summary ([classname])`
	Collects a summary of object counts into a table and returns this table as a string. The table shows the number of objects in each specified class. For example:
	`3 Array`
	`5 String`

classname	Optional. The type of object to count. If not supplied, counts all objects currently defined.
write `$.write (text[, text...]...)`	Writes the specified text to the JavaScript Console. Returns `undefined`.
text	One or more strings to write, which are concatenated to form a single string.
writeln `$.writeln (text[, text...]...)`	Writes the specified text to the JavaScript Console and appends a linefeed sequence. Returns `undefined`.
text	One or more strings to write, which are concatenated to form a single string.

Object statistics

The output from `$.list()` is formatted as in the following example.

Address	L	Refs	Prop	Class	Name
0092196c	4	0	Function	[toplevel]	
00976c8c	2	1	Object	Object	
00991bc4 L	1	1	LOTest	LOTest	
0099142c L	2	2	Function	LOTest	
00991294	1	0	Object	Object	workspace

The columns show the following object information.

Address	The physical address of the object in memory.
L	This column contains the letter "L" if the object is a LiveObject (which is an internal data type).
Refs	The reference count of the object.
Prop	A second reference count for the number of properties that reference the object. The garbage collector uses this count to break circular references. If the reference count is not equal to the number of JavaScript properties that reference it, the object is considered to be used elsewhere and is not garbage collected.
Class	The class name of the object.
Name	The name of the object. This name does not reflect the name of the property the object has been stored into. The name is mostly relevant to Function objects, where it is the name of the function or method. Names in brackets are internal names of scripts. If the object has an ID, the last column displays that ID.

ExtendScript Reflection Interface

ExtendScript provides a reflection interface that allows you to find out everything about an object, including its name, a description, the expected data type for properties, the arguments and return value for methods, and any default values or limitations to the input values.

Reflection Object

Every object has a `reflect` property that returns a `reflection` object that reports the contents of the object. You can, for example, show the values of all the properties of an object with code like this:

```
var f= new File ("myfile");
var props = f.reflect.properties;
for (var i = 0; i < props.length; i++) {
    $.writeln('this property ' + props[i].name + ' is ' + f[props[i].name]);
}
```

Reflection object properties

All properties are read only.

description	String	Short text describing the reflected object, or `undefined` if no description is available.
help	String	Longer text describing the reflected object more completely, or `undefined` if no description is available.
methods	Array of ReflectionInfo	An Array of ReflectionInfo Objects containing all methods of the reflected object, defined in the class or in the specific instance.
name	String	The class name of the reflected object.
properties	Array of ReflectionInfo	An Array of ReflectionInfo Objects containing all properties of the reflected object, defined in the class or in the specific instance. For objects with dynamic properties (defined at runtime) the list contains only those dynamic properties that have already been accessed by the script. For example, in an object wrapping an HTML tag, the names of the HTML attributes are determined at run time.

Reflection object functions

find *reflectionObj*.find (*name*)	Returns the ReflectionInfo Object for the named property of the reflected object, or `null` if no such property exists. Use this method to get information about dynamic properties that have not yet been accessed, but that are known to exist.
name	The property for which to retrieve information.

➤ Examples

This code determines the class name of an object:

```
obj = new String ("hi");
obj.reflect.name; // => String
```

This code gets a list of all methods:

```
obj = new String ("hi");
obj.reflect.methods; //=> indexOf,slice,...
obj.reflect.find ("indexOf"); // => the method info
```

This code gets a list of properties:

```
Math.reflect.properties; //=> PI,LOG10,...
```

This code gets the data type of a property:

```
Math.reflect.find ("PI").type; // => number
```

ReflectionInfo Object

This object contains information about a property, a method, or a method argument.

- You can access `ReflectionInfo` objects in a Reflection Object's `properties` and `methods` arrays, by name or index:

  ```
  obj = new String ("hi");
  obj.reflect.methods[0];
  obj.reflect.methods["indexOf"];
  ```

- You can access the `ReflectionInfo` objects for the arguments of a method in the `arguments` array of the `ReflectionInfo` object for the method, by index:

  ```
  obj.reflect.methods["indexOf"].arguments[0];
  ```

ReflectionInfo object properties

`arguments`	Array of ReflectionInfo	For a reflected method, an array of ReflectionInfo Objects describing each method argument.

`dataType`	String	The data type of the reflected element. One of:
		`boolean`
		`number`
		`string`
		`Classname`: The class name of an object.
		Note: Class names start with a capital letter. Thus, the value `string` stands for a JavaScript string, while `String` is a JavaScript `String` wrapper object.
		`*`: Any type. This is the default.
		`null`
		`undefined`: Return data type for a function that does not return any value.
		`unknown`

`defaultValue`	any	The default value for a reflected property or method argument, or `undefined` if there is no default value, if the property is undefined, or if the element is a method.
`description`	String	Short text describing the reflected object, or `undefined` if no description is available.

help	String	Longer text describing the reflected object more completely, or `undefined` if no description is available.
isCollection	Boolean	When `true`, the reflected property or method returns a collection; otherwise, `false`.
max	Number	The maximum numeric value for the reflected element, or `undefined` if there is no maximum or if the element is a method.
min	Number	The minimum numeric value for the reflected element, or `undefined` if there is no minimum or if the element is a method.
name	String Number	The name of the reflected element. A string, or a number for an array index.
type	String	The type of the reflected element. One of: `readonly`: A read-only property. `readwrite`: A read-write property. `createonly`: A property that is valid only during creation of an object. `method`: A method.

Localizing ExtendScript Strings

Localization is the process of translating and otherwise manipulating an interface so that it looks as if it had been originally designed for a particular language. ExtendScript gives you the ability to localize the strings in your script's user interface. The language is chosen by the application at startup, according to the current locale provided by the operating system.

For portions of your user interface that are displayed on the screen, you may want to localize the displayed text. You can localize any string explicitly using the Global localize function, which takes as its argument a *localization object* containing the localized versions of a string.

A localization object is a JavaScript object literal whose property names are locale names, and whose property values are the localized text strings. The locale name is a standard language code with an optional region identifier. For details of the syntax, see Locale names.

In this example, a `msg` object contains localized text strings for two locales. This object supplies the text for an alert dialog.

```
msg = { en: "Hello, world", de: "Hallo Welt" };
alert (msg);
```

ExtendScript matches the current locale and platform to one of the object's properties and uses the associated string. On a German system, for example, the property `de: "Hallo Welt"` is converted to the string `"Hallo Welt"`.

Variable values in localized strings

Some localization strings need to contain additional data whose position and order may change according to the language used.

You can include variables in the string values of the localization object, in the form `%n`. The variables are replaced in the returned string with the results of JavaScript expressions, supplied as additional arguments to the `localize` function. The variable `%1` corresponds to the first additional argument, `%2` to the second, and so on.

Because the replacement occurs after the localized string is chosen, the variable values are inserted in the correct position. For example:

```
today = {
   en: "Today is %1/%2.",
   de: "Heute ist der %2.%1."
   };
d = new Date();
alert (localize (today, d.getMonth()+1, d.getDate()));
```

Enabling automatic localization

ExtendScript offers an automatic localization feature. When it is enabled, you can specify a localization object directly as the value of any property that takes a localizable string, without using the `localize` function. For example:

```
msg = { en: "Yes", de: "Ja", fr: "Oui" };
alert (msg);
```

To use automatic translation of localization objects, you must enable localization in your script with this statement:

```
$.localize = true;
```

The `localize` function always performs its translation, regardless of the setting of the `$.localize` variable. For example:

```
msg = { en: "Yes", de: "Ja", fr: "Oui" };
//Only works if the $.localize=true
alert (msg);
//Always works, regardless of $.localize value
alert ( localize (msg));
```

If you need to include variables in the localized strings, use the `localize` function.

Locale names

A locale name is an identifier string in that contains an ISO 639 language specifier, and optionally an ISO 3166 region specifier, separated from the language specifier by an underscore.

- The ISO 639 standard defines a set of two-letter language abbreviations, such as `en` for English and `de` for German.
- The ISO 3166 standard defines a region code, another two-letter identifier, which you can optionally append to the language identifier with an underscore. For example, `en_US` identifies U.S. English, while `en_GB` identifies British English.

This object defines one message for British English, another for all other flavors of English, and another for all flavors of German:

```
message = {
  en_GB: "Please select a colour."
  en: "Please select a colour."
  de: "Bitte wählen Sie eine Farbe."
};
```

If you need to specify different messages for different platforms, you can append another underline character and the name of the platform, one of `Win`, `Mac`, or `Unix`. For example, this objects defines one message in British English to be displayed in Mac OS, one for all other flavors of English in Mac OS, and one for all other flavors of English on all other platforms:

```
pressMsg = {
  en_GB_Mac: "Press Cmd-S to select a colour.",
  en_Mac: "Press Cmd-S to select a color.",
  en: "Press Ctrl-S to select a color."
};
```

All of these identifiers are case sensitive. For example, `EN_US` is not valid.

➤ How locale names are resolved

1. ExtendScript gets the hosting application’s locale; for example, `en_US`.
2. It appends the platform identifier; for example, `en_US_Win`.
3. It looks for a matching property, and if found, returns the value string.
4. If not found, it removes the platform identifier (for example, `en_US`) and retries .

5. If not found, it removes the region identifier (for example, `en`) and retries .
6. If not found, it tries the identifier `en` (that is, the default language is English).
7. If not found, it returns the entire localizer object.

Testing localization

ExtendScript stores the current locale in the variable `$.locale`. This variable is updated whenever the locale of the hosting application changes.

To test your localized strings, you can temporarily reset the locale. To restore the original behavior, set the variable to `null`, `false`, 0, or the empty string. An example:

```
$.locale = "ru"; // try your Russian messages
$.locale = null; // restore to the locale of the app
```

Global localize function

The globally available `localize` function can be used to provide localized strings anywhere a displayed text value is specified.

`localize`

`localize (localization_obj[, args])`
`localize (ZString)`

Returns the localized string for the current locale.

localization_obj	A JavaScript object literal whose property names are locale names, and whose property values are the localized text strings. The locale name is an identifier as specified in the ISO 3166 standard, a set of two-letter language abbreviations, such as "`en`" for English and "`de`" for German.
	For example:
	`btnText = { en: "Yes", de: "Ja", fr: "Oui" };`
	`b1 = w.add ("button", undefined, localize (btnText));`
	The string value of each property can contain variables in the form `%1`, `%2`, and so on, corresponding to additional arguments. The variable is replaced with the result of evaluating the corresponding argument in the returned string.

args	Optional. Additional JavaScript expressions matching variables in the string values supplied in the localization object. The first argument corresponds to the variable `%1`, the second to `%2`, and so on. Each expression is evaluated and the result inserted in the variable's position in the returned string.

➤ **Example**

```
today = {
  en: "Today is %1/%2",
  de: "Heute ist der %2.%1."
  };
d = new Date();
alert (localize (today, d.getMonth()+1, d.getDate()));
```

ZString	Internal use only. A ZString is an internal Adobe format for localized strings, which you might see in Adobe scripts. It is a string that begins with `$$$` and contains a path to the localized string in an installed ZString dictionary. For example: `w = new Window ("dialog", localize ("$$$/UI/title1=Sample"));`

User Notification Helper Functions

ExtendScript provides a set of globally available functions that allow you to display short messages to the user in platform-standard dialog boxes. There are three types of message dialogs:

- **Alert**: Displays a dialog containing a short message and an **OK** button.
- **Confirm**: Displays a dialog containing a short message and two buttons, **Yes** and **No**, allowing the user to accept or reject an action.
- **Prompt**: Displays a dialog containing a short message, a text entry field, and **OK** and **Cancel** buttons, allowing the user to supply a value to the script.

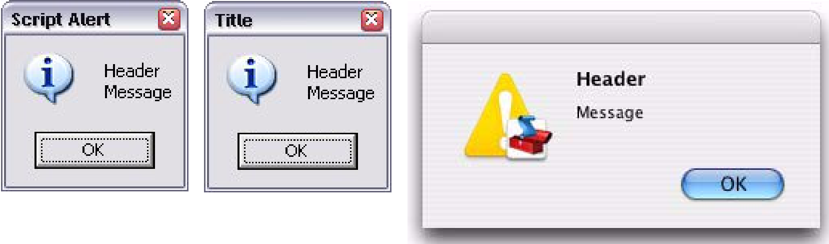
These dialogs are customizable to a small degree. The appearance is platform specific.

Global alert function

`alert` `alert (message[, title, errorIcon]);`	Displays a platform-standard dialog containing a short message and an **OK** button. Returns `undefined`.
`message`	The string for the displayed message.
`title`	Optional. A string to appear as the title of the dialog, if the platform supports a title. Mac OS does not support titles for alert dialogs. The default title string is "Script Alert".
`errorIcon`	Optional. When `true`, the platform-standard alert icon is replaced by the platform-standard error icon in the dialog. Default is `false`.

➤ Examples

This figure shows simple alert dialogs in Windows and Mac OS.



This figure shows alert dialogs with error icons.

Global confirm function

confirm `confirm (message[,noAsDflt ,title ]);`	Displays a platform-standard dialog containing a short message and two buttons labeled **Yes** and **No**. Returns `true` if the user clicked **Yes**, `false` if the user clicked **No**.
message	The string for the displayed message.
noAsDflt	Optional. When `true`, the **No** button is the default choice, selected when the user types ENTER. Default is `false`, meaning that **Yes** is the default choice.
title	Optional. A string to appear as the title of the dialog, if the platform supports a title. Mac OS does not support titles for confirmation dialogs. The default title string is "Script Alert".

➤ Examples

This figure shows simple confirmation dialogs in Windows and Mac OS.

This figure shows confirmation dialogs with **No** as the default button.

Global prompt function

prompt `prompt (message, preset[, title ]);`	Displays a platform-standard dialog containing a short message, a text edit field, and two buttons labeled **OK** and **Cancel**. Returns the value of the text edit field if the user clicked **OK**, `null` if the user clicked **Cancel**.
message	The string for the displayed message.
preset	The initial value to be displayed in the text edit field.
title	Optional. A string to appear as the title of the dialog. In Windows, this appears in the window's frame, while in Mac OS it appears above the message. The default title string is "Script Prompt".

➤ Examples

This figure shows simple prompt dialogs in Windows and Mac OS.

This figure shows confirmation dialogs with a `title` value specified.

Specifying Measurement Values

ExtendScript provides the UnitValue Object to represent measurement values. The properties and methods of the `UnitValue` object make it easy to change the value, the unit, or both, or to perform conversions from one unit to another.

UnitValue Object

Represents measurement values that contain both the numeric magnitude and the unit of measurement.

UnitValue object constructor

The `UnitValue` constructor creates a new `UnitValue` object. The keyword `new` is optional:

```
myVal = new UnitValue (value, unit);
myVal = new UnitValue ("value unit");
myVal = new UnitValue (value, "unit");
```

The *value* is a number, and the *unit* is specified with a string in abbreviated, singular, or plural form, as shown in the following table.

Abbreviation	Singular	Plural	Comments
in	inch	inches	2.54 cm
ft	foot	feet	30.48 cm
yd	yard	yards	91.44 cm
mi	mile	miles	1609.344 m
mm	millimeter	millimeters	
cm	centimeter	centimeters	
m	meter	meters	
km	kilometer	kilometers	
pt	point	points	inches / 72
pc	pica	picas	points * 12
tpt	traditional point	traditional points	inches / 72.27
tpc	traditional pica	traditional picas	12 tpt
ci	cicero	ciceros	12.7872 pt
px	pixel	pixels	baseless (see below)
%	percent	percent	baseless (see below)

If an unknown unit type is supplied, the type is set to `"?"`, and the `UnitValue` object prints as "UnitValue 0.00000".

For example, all of the following formats are equivalent:

```
myVal = new UnitValue (12, "cm");
```

```
myVal = new UnitValue ("12 cm");
myVal = UnitValue ("12 centimeters");
```

UnitValue object properties

baseUnit	UnitValue	A UnitValue Object that defines the size of one pixel, or a total size to use as a base for percentage values. This is used as the base conversion unit for pixels and percentages; see Converting pixel and percentage values. Default is 0.013889 inches (1/72 in), which is the base conversion unit for pixels at 72 dpi. Set to `null` to restore the default.
type	String	The unit type in abbreviated form; for example, "cm" or "in".
value	Number	The numeric measurement value.

UnitValue object functions

as *unitValueObj*.as (*unit*)	Returns the numeric value of this object in the given unit. If the unit is unknown or cannot be computed, generates a run-time error.
unit	The unit type in abbreviated form; for example, "cm" or "in".
convert *unitValueObj*.convert (*unit*)	Converts this object to the given unit, resetting the `type` and `value` accordingly. Returns `true` if the conversion is successful. If the unit is unknown or the object cannot be converted, generates a run-time error and returns `false`.
unit	The unit type in abbreviated form; for example, "cm" or "in".

Converting pixel and percentage values

Converting measurements among different units requires a common base unit. For example, for length, the meter is the base unit. All length units can be converted into meters, which makes it possible to convert any length unit into any other length unit.

Pixels and percentages do not have a standard common base unit. Pixel measurements are relative to display resolution, and percentages are relative to an absolute total size.

- To convert pixels into length units, you must know the size of a single pixel. The size of a pixel depends on the display resolution. A common resolution measurement is 72 dpi, which means that there are 72 pixels to the inch. The conversion base for pixels at 72 dpi is 0.013889 inches (1/72 inch).
- Percentage values are relative to a total measurement. For example, 10% of 100 inches is 10 inches, while 10% of 1 meter is 0.1 meters. The conversion base of a percentage is the unit value corresponding to 100%.

The default `baseUnit` of a `unitValue` object is 0.013889 inches, the base for pixels at 72 dpi. If the `unitValue` is for pixels at any other dpi, or for a percentage value, you must set the `baseUnit` value accordingly. The `baseUnit` value is itself a `unitValue` object, containing both a magnitude and a unit.

For a system using a different dpi, you can change the `baseUnit` value in the `UnitValue` class, thus changing the default for all new `unitValue` objects. For example, to double the resolution of pixels:

```
UnitValue.baseUnit = UnitValue (1/144, "in"); //144 dpi
```

To restore the default, assign `null` to the class property:

```
UnitValue.baseUnit = null; //restore default
```

You can override the default value for any particular `unitValue` object by setting the property in that object. For example, to create a unitValue object for pixels with 96 dpi:

```
pixels = UnitValue (10, "px");
myPixBase = UnitValue (1/96, "in");
pixels.baseUnit = myPixBase;
```

For percentage measurements, set the `baseUnit` property to the measurement value for 100%. For example, to create a `unitValue` object for 40 % of 10 feet:

```
myPctVal = UnitValue (40, "%");
myBase = UnitValue (10, "ft")
myPctVal.baseUnit = myBase;
```

Use the as method to get to a percentage value as a unit value:

```
myFootVal = myPctVal.as ("ft"); // => 4
myInchVal = myPctVal.as ("in"); // => 36
```

You can convert a `unitValue` from an absolute measurement to pixels or percents in the same way:

```
myMeterVal = UnitValue (10, "m"); // 10 meters
myBase = UnitValue (1, "km");
myMeterVal.baseUnit = myBase; //as a percentage of 1 kilometer
pctOfKm = myMeterVal.as ('%'); // => 1

myVal = UnitValue ("1 in"); // Define measurement in inches
// convert to pixels using default base
myVal.convert ("px"); // => value=72 type=px
```

Computing with unit values

`UnitValue` objects can be used in computational JavaScript expressions. The way the value is used depends on the type of operator.

- Unary operators (`~`, `!`, `+`, `-`)

`~unitValue`	The numeric value is converted to a 32-bit integer with inverted bits.
`!unitValue`	Result is `true` if the numeric value is nonzero, `false` if it is not.
`+unitValue`	Result is the numeric value.
`-unitValue`	Result is the negated numeric value.

- Binary operators (`+`, `-`, `*`, `/`, `%`)

  If one operand is `unitValue` object and the other is a number, the operation is applied to the number and the numeric value of the object. The expression returns a new `unitValue` object with the result as its `value`. For example:

```
val = new UnitValue ("10 cm");
res = val * 20;
// res is a UnitValue (200, "cm");
```

  If both operands are `unitValue` objects, JavaScript converts the right operand to the same unit as the left operand and applies the operation to the resulting values. The expression returns a new `unitValue` object with the unit of the left operand, and the result `value`. For example:

```
a = new UnitValue ("1 m");
b = new UnitValue ("10 cm");
a + b;
// res is a UnitValue (1.1, "m");
b + a;
// res is a UnitValue (110, "cm");
```

- Comparisons (`=`, `==`, `<`, `>`, `<=`, `>=`)

  If one operand is a `unitValue` object and the other is a number, JavaScript compares the number with the `unitValue`'s numeric value.

  If both operands are `unitValue` objects, JavaScript converts both objects to the same unit, and compares the converted numeric values.

  For example:

```
a = new UnitValue ("98 cm");
b = new UnitValue ("1 m");
a < b;   // => true
a < 1;   // => false
a == 98; // => true
```

Modular Programming Support

ExtendScript provides support for a modular approach to scripting by allowing you to include one script in another as a resource, and allowing a script to export definitions that can be imported and used in another script.

Preprocessor directives

ExtendScript provides preprocessor directives for including external scripts, naming scripts, specifying an ExtendScript engine, and setting certain flags. You can specify these in two ways:

- With a C-style statement starting with the # character:

```
#include "file.jsxinc"
```

- In a comment whose text starts with the @ character:

```
// @include "file.jsxinc"
```

When a directive takes one or more arguments, and an argument contains any nonalphanumeric characters, the argument must be enclosed in single or double quotes. This is generally the case with paths and file names, for example, which contain dots and slashes.

#engine *name*	Identifies the ExtendScript engine that runs this script. This allows other engines to refer to the scripts in this engine by importing the exported functions and variables. See Importing and exporting between scripts.
	Use JavaScript identifier syntax for the name. Enclosing quotes are optional. For example:
	`#engine library`
	`#engine "$lib"`

#include *file*	Includes a JavaScript source file from another location. Inserts the contents of the named file into this file at the location of this statement. The *file* argument is an Adobe portable file specification. See Specifying Paths. As a convention, use the file extension `.jsxinc` for JavaScript include files. For example: `#include "../include/lib.jsxinc"` To set one or more paths for the `#include` statement to scan, use the `#includepath` preprocessor directive. If the file to be included cannot be found, ExtendScript throws a run-time error. Included source code is not shown in the debugger, so you cannot set breakpoints in it.

<table>
<tr><td><code>#includepath path</code></td><td>
One or more paths that the <code>#include</code> statement should use to locate the files to be included. The semicolon (<code>;</code>) separates path names.
If a <code>#include</code> file name starts with a slash (<code>/</code>), it is an absolute path name, and the include paths are ignored. Otherwise, ExtendScript attempts to find the file by prefixing the file with each path set by the <code>#includepath</code> statement.
For example:
<pre>#includepath "include;../include"
#include "file.jsxinc"</pre>
Multiple <code>#includepath</code> statements are allowed; the list of paths changes each time an <code>#includepath</code> statement is executed.
As a fallback, ExtendScript also uses the contents of the environment variable <code>JSINCLUDE</code> as a list of include paths.
Some engines can have a predefined set of include paths. If so, the path provided by <code>#includepath</code> is tried before the predefined paths. If, for example, the engine has a predefined path set to <code>predef;predef/include</code>, the preceding example causes the following lookup sequence:
<code>file.jsxinc</code>: literal lookup
<code>include/file.jsxinc</code>: first <code>#includepath</code> path
<code>../include/file.jsxinc:</code> second <code>#includepath</code> path
<code>predef/file.jsxinc</code>: first predefined engine path
<code>predef/include/file.jsxinc</code>: second predefined engine path
</td></tr>
<tr><td><code>#script name</code></td><td>
Names a script. Enclosing quotes are optional, but required for names that include spaces or special characters. For example:
<pre>#script SetupPalette
#script "Load image file"</pre>
The <code>name</code> value is displayed in the Toolkit Editor tab. An unnamed script is assigned a unique name generated from a number.
</td></tr>
<tr><td><code>#strict on</code></td><td>Turns on strict error checking. See the Dollar ($) Object's strict property.</td></tr>
<tr><td><code>#target name</code></td><td>
Defines the target application of this JSX file. The <code>name</code> value is an application specifier; see Application and Namespace Specifiers. Enclosing quotes are optional.
If the Toolkit is registered as the handler for files with the <code>.jsx</code> extension (as it is by default), opening the file opens the target application to run the script. If this directive is not present, the Toolkit loads and displays the script. A user can open a file by double-clicking it in a file browser, and a script can open a file using a <code>File</code> object's <code>execute</code> method.
</td></tr>
</table>

Importing and exporting between scripts

The ExtendScript JavaScript language has been extended to support function calls and variable access across various source code modules and ExtendScript engines. A script can use the `export` statement to make its definitions available to other scripts, which use the `import` statement to access those definitions.

To use this feature, the exporting script must name its ExtendScript engine using the `#engine` preprocessor statement. The name must follow JavaScript naming syntax; it cannot be an expression.

For example, the following script could serve as a library or resource file. It defines and exports a constant and a function:

```
#engine library
export random, libVersion;
const libVersion = "Library 1.0";
function random (max) {
   return Math.floor (Math.random() * max);
}
```

A script running in a different engine can import the exported elements. The import statement identifies the resource script that exported the variables using the engine name:

```
import library.random, library.libVersion;
print (random (100));
```

You can use the asterisk wildcard (`*`) to import all symbols exported by a library:

```
import library.*
```

Objects cannot be transferred between engines. You cannot retrieve or store objects, and you cannot call functions with objects as arguments. However, you can use the JavaScript `toSource` function to serialize objects into strings before passing them. You can then use the JavaScript `eval` function to reconstruct the object from the string.

For example, this function takes as its argument a serialized string and constructs an object from it:

```
function myFn (serialized) {
   var obj = eval (serialized);
   // continue working...
}
```

In calling the function, you deconstruct the object you want to pass into a serialized string:

```
myFn (myObject.toSource()); // pass a serialized object
```

Operator Overloading

ExtendScript allows you to extend or override the behavior of a math or a Boolean operator for a specific class by defining a method in that class with same name as the operator. For example, this code defines the addition (+) operator for the class `MyClass`. In this case, the addition operator simply adds the operand to the property value:

```
// define the constructor method
function MyClass (initialValue) {
   this.value = initialValue;
}
// define the addition operator
MyClass.prototype ["+"] = function (operand) {
   return this.value + operand;
}
```

This allows you to perform the "+" operation with any object of this class:

```
var obj = new MyClass (5);
Result: [object Object]
obj + 10;
Result: 15
```

You can override the following operators:

Unary	`+, -` `~`

Binary	`+, -`
	`*, /, %, ^`
	`<, <=, ==`
	`<<, >>, >>>`
	`&, |, ===`

- The operators `>` and `>=` are implemented by executing NOT operator `<=` and NOT operator `<`.
- Combined assignment operators such as `*=` are not supported.

All operator overload implementations must return the result of the operation. To perform the default operation, return `undefined`.

Unary operator functions work on the `this` object, while binary operators work on the `this` object and the first argument. The + and - operators have both unary and binary implementations. If the first argument is undefined, the operator is unary; if it is supplied, the operator is binary.

For binary operators, a second argument indicates the order of operands. For noncommutative operators, either implement both order variants in your function or return `undefined` for combinations that you do not support. For example:

```
this ["/"] = function (operand, rev) {
   if (rev) {
      // do not resolve operand / this
      return;
   } else {
      // resolve this / operand
      return this.value / operand;
   }
}
```

Application and Namespace Specifiers

All forms of interapplication communication use Application specifiers to identify Adobe applications.

- In all ExtendScript scripts, the `#target` directive can use an specifier to identify the application that should run that script. See Preprocessor directives.
- In interapplication messages, the specifier is used as the value of the `target` property of the message object, to identify the target application for the message.
- Bridge (which is integrated with all ExtendScript-enabled applications) uses an application specifier as the value of the `document.owner` property, to identify another Creative Suite 2 application that created or opened a Bridge browser window. For details, see the Document Object.

When a script for one application invokes Cross-DOM or exported functions, it identifies the exporting application using Namespace specifiers.

Application specifiers

Application specifiers are strings that encode the application name, a version number and a language code. They take the following form:

appname[-*version*[-*locale*]]

appname	An Adobe application name. One of:
	`acrobat`
	`aftereffects`
	`atmosphere`
	`audition`
	`bridge`
	`encore`
	`golive`
	`illustrator`
	`incopy`
	`indesign`
	`photoshop`
	`premiere`

version	Optional. A number indicating at least a major version. If not supplied, the most recent version is assumed. The number can include a minor version separated from the major version number by a dot; for example, `1.5`.
locale	Optional. An Adobe locale code, consisting of a 2-letter ISO-639 language code and an optional 2-letter ISO 3166 country code separated by an underscore. Case is significant. For example, `en_US`, `en_UK`, `ja_JP`, `de_DE`, `fr_FR`. If not supplied, ExtendScript uses the current platform locale. Do not specify a locale for a multilingual application, such as Bridge, that has all locale versions included in a single installation.

The following are examples of legal specifiers:

```
photoshop
bridge-1
bridge-1.0
illustrator-12.2
bridge-1-en_us
golive-8-de_de
```

Namespace specifiers

When calling cross-DOM and exported functions from other applications, a namespace specifier qualifies the function call, directing it to the appropriate application.

Namespace specifiers consist of an application name, as used in an application specifier, with an optional major version number. Use it as a prefix to an exported function name, with the JavaScript dot notation.

```
appname[majorVersion].functionName(args)
```

For example:

- To call the cross-DOM function `quit` in Photoshop CS2, use `photoshop.quit()`, and to call it in GoLive CS2, use `golive.quit()`.
- To call the exported function `place`, defined for Illustrator CS version 12, call `illustrator12.place(myFiles)`.

For information about the cross-DOM and exported functions, see Interapplication Communication with Scripts.

Script Locations and Checking Application Installation

On startup, all ExtendScript-enabled applications execute all JSX files that they find in the user startup folder:

- In Windows, the startup folder is:

  ```
  %APPDATA%\Adobe\StartupScripts
  ```
- In Mac OS, the startup folder is

  ```
  ~/Library/Application Support/Adobe/StartupScripts/
  ```

A script in the startup directory is executed on startup by all applications. If you place a script here, it must contain code to check whether it is being run by the intended application. You can do this using the `appName` static property of the `BridgeTalk` class. For example:

```
if( BridgeTalk.appName == "bridge" ) {
      //continue executing script
   }
```

In addition, individual applications may look for application-specific scripts in a subfolder named with that application's specifier and version, in the form:

```
%APPDATA%\Adobe\StartupScripts\appname\version
~/Library/Application Support/Adobe/StartupScripts/appname/version/
```

The name and version in these folder names are specified in the form required for Application specifiers. For example, in Windows, GoLive CS2 version 8.2 would look for scripts in the directory:

```
%APPDATA%\Adobe\StartupScripts\golive\8.2
```

The *version* portion of the Bridge-specific folder path is an exact version number. That is, scripts in the folder `bridge/1.5` are executed only by Bridge version 1.5, and so on.

Individual applications may also implement a path in the installation directory for application-specific startup scripts. For example:

```
IllustratorCS2_install_dir\Startup Scripts
```

```
IllustratorCS2_install_dir/Startup Scripts/
```

If a script that is run by one application will communicate with another application, or add functionality that depends on another application, it must first check whether that application and version is installed. You can do this using the `BridgeTalk.getSpecifier` static function. For example:

```
if( BridgeTalk.appName == "bridge" ) {
// Check that PS CS2 is installed
   if( BridgeTalk.getSpecifier("photoshop",9)){
      // add PS automate menu to Bridge UI
   }
}
```

For details of interapplication communication, see Interapplication Communication with Scripts.

Index

A

absolute pathnames 56
Adobe Bridge
  document object model (DOM) 23
  events 31
  global information 94
  quitting programmatically 97
  UI elements 28
  windows 24
alerts 69, 147, 226
aliases, referencing 59
application
  message framework 184, 193
  preferences 110, 119, 122
  specifiers 237
application object
  about 24
  registering event handlers 32
applications
  accessing from other applications 194
  as script execution targets 234
  calling exported functions 238
  communication between 180, 237
  debugging 205
  launching to open files 127
arrays, passing between engines 51
automatic layout of UI controls 75, 168

B

backslashes in pathnames 57
breakpoints 212
Bridge, *See* Adobe Bridge
BridgeTalk object
  See also interapplication communication
  global values for messaging framework 193
  message objects 197
browse schemes
  defining and registering 36, 97
  predefined and script-defined 36, 123
browser window
  as document object 23, 100
  opening 95
  panels 25
  parts 28
button objects 66, 154

C

call stack for execution 208
callback and execJS interaction 53
callbacks
  defining for HTML scripts in Content pane 100
  defining for HTML scripts in dialogs 99
  defining for HTML scripts in nav bars 116
  defining for remote calls between HTML scripts and Bridge 42
  executing 104, 118
  scheduling execution of remote function 97
character encoding 59
checkbox objects 67, 154
command line
  for JavaScript 207
  for platform 97
commands
  adding to menus 169
  identifiers 173
confirmation dialogs 69, 147, 226
console 207
containers for UI controls 65
Content pane
  about 29
  how thumbnails are displayed 36, 123
  using to display a user interface 39, 50
context menus 30
control types for ScriptUI windows 65
cross-DOM
  about 181
  function reference 181
  specifying application 238

D

data
  passing between applications 190
  passing between engines 42, 51
  tracking while debugging 206
debugging
  call stack 208
  in ExtendScript Toolkit 210
  optimization tools 213
  selecting target application 205
  setting breakpoints 212
debugging tools
  Dollar ($) object 215
  ExtendScript Toolkit 203
  Reflection object 219
dialogs
  about 26, 39
  callbacks and remote function execution 44
  closing 44
  creating with ScriptUI 61
  displaying HTML controls 44, 98
  displaying ScriptUI 40
  modal and modeless
    HTML 44
    ScriptUI 61, 69
  predefined 147
  Preferences 110, 119, 122

directories, referencing 138
directory specifications 56
displaying a user interface for a script 38
document object 100
document object model (DOM) 23
documents
about 23
and the application object 94
opening 95
user interaction events 108
Dollar ($) object 215
drives, specifying in paths 58
dropdownlist objects 68, 155

E

edittext objects 66, 155
embedded browser for displaying HTML in Bridge 42
encoding 59
engine ExtendScript directive 233
engines, JavaScript 42, 205
error handling
filesystem 60, 143
messaging framework 201
setting strict 234
event handling
Bridge events 31
defining handlers 31
event object argument to handlers 106
registering handlers 32
user interface events 35
events
about 27
event objects passed to handlers 106
global 107
in Browser window 108
in Preferences dialog 110, 122
in ScriptUI windows 73
in thumbnails 109
exchanging data between engines 42, 51
executing JavaScript
about 22, 238
in ExtendScript Toolkit 203, 210
executing remote functions
scheduling from callback 53
with execJS 43
execution call stack 208
exiting the Bridge application 97
exported functions
about 180
specifying application 238
exporting and importing scripts 234
ExtendScript
command line 207
Dollar ($) object 215
engines 233
multiple engines 205
operator overloading 236
preprocessor directives 233
Reflection object 219
ScriptUI module 61, 146
ExtendScript Toolkit 203
configuring window 204
debugging 210
editing scripts 209
optimization tools 213
setting breakpoints 212

F

Favorites pane
about 28
adding nodes 36
object 111
File object 129
files
distinguishing from folders 56
loading local copies of VersionCue references 96
metadata 113
name and path specifications 56
opening from thumbnail object 127
files and folders
how they are displayed in Content pane 36, 123
platform-independent reference objects 56
filesystem
aliases 59
error handling 60, 143
object references 56, 129
filtering web pages 103
flyout menus 30
Folder object 138
Folders pane 29
frames for UI controls 62, 65
functions
call stack for debugging 208
calling in other applications 180
cross-DOM 181
global 95
operating system 97

G

global application events 107
global dialogs 69, 147, 226
global functions 95
global information 24, 94
global localize function 225
global object 215
GoLive
more information 21
grouping controls 65, 156

H

handlers
for Bridge events 31
for ScriptUI events 73
registering 32
HTML pages, displaying in Content pane 100, 125
HTML script functions, executing remotely 43
HTML user interfaces

about 38, 42
callbacks for remote calls between HTML scripts and Bridge 42
displaying in Content pane 50
displaying in dialogs 98
displaying in nav bar 47
running in browser and Bridge engines 42

I

I/O, unicode 59
iconbutton objects 66, 156
icons, displaying in ScriptUI windows 68
image file metadata 113
image objects 67, 156
importing and exporting scripts 234
include ExtendScript directive 233
includepath ExtendScript directive 234
interaction with users 26
interapplication communication
about 23, 180, 237
checking application installation 194, 239
cross-DOM functions 181
message objects 197
messaging 184, 193
specifying target applications 238
internationalization
ExtendScript utilities 222
in ScriptUI windows 91
item objects 68, 156

J

JavaScript
console 207
debugging 203
editing scripts 209
engines 233
more information 21
multiple engines 205
sending to other applications 184
JavaScript engines for web browser and Bridge 42

K

Keywords pane 29

L

labels, thumbnail 126
layout of user interface controls 62, 75, 168
layout properties in ScriptUI windows 76
listbox objects 68, 157
listitem objects 68
local copies of referenced VersionCue files 96
locale identifiers 223
localization
ExtendScript utilities 222
in ScriptUI windows 91
localize function 225
locations of startup scripts 22, 238

M

main window
as document object 23
opening 95
parts 28
measurement values 229
menu element objects 169
menu items, *See* commands
menubar
about 30
extending 169
menus
about 27
context and flyout 30
extending 169
identifiers 172
messages
handling responses 187
passing data 190
receiving 186
responding to 186
sending 184
messaging
about 180
error handling 201
framework 184, 193
global values in BridgeTalk object 193
message objects 197
specifying applications 237
metadata
display 29
example of access through HTML UI 50
object 113
modal and modeless dialogs
HTML 44
ScriptUI 61, 69

N

namespaces
for external functions 180
global 215
specifiers for interapplication communication 238
naming scripts 234
navigation bars
about 26, 38
accessing 101
displaying HTML 47
displaying ScriptUI 40
executing remote functions in 48
object 116
using callbacks 47
nodes
about 123
accessing 125
notifications 226

O

object model
application and document 24

correspondance to Bridge UI 28
thumbnails 25
objects
about the object model 23
application 94
creation properties 64
document 100
file and folder reference 56, 129, 138
global object 215
menus and commands 169
message 197
navigation bars 116
passing between engines 51
preferences 119, 122
retrieving information about 219
thumbnail 123
user interface 61, 146
user interface controls 154
windows 147
operating system, access to commands 97
operator overloading in ExtendScript 236
optimization tools 213

P

palettes 61
panel objects 62, 65, 157
passing values between engines 42, 51
path specifications 56
pathnames
separator characters 57
special characters 57
placing windows and controls 166
platform commands 97
platform-independent paths 56
portability of file references 59
preferences
about 27
dialog object 122
events 110
object 119
Preview pane 29
profiling for script optimization 213
progressbar objects 67, 157
prompts 69, 148, 226

Q

quitting the Bridge application 97

R

radiobutton objects 67, 158
redirecting URLs 103
Reflection object 219
relative pathnames 56
remote function execution 43
resource strings for ScriptUI elements 71
responding to user input 38
responding to user interaction with Bridge 35, 106

S

scheduling execution after return from callback 97
scheduling tasks 53
script execution, setting target application 234
script ExtendScript directive 234
scripting overview 22
scripts
access to Bridge window 28
checking application installation 194, 239
command line 207
debugging 203
editing in ExtendScript Toolkit 209
executing 22, 238
importing and exporting definitions 234
including in other scripts 233
naming 234
output 207
sending to other applications 184
startup 22, 238
user interface options 38
ScriptUI
about 61
control types 65
in Bridge windows 40
layout properties 76
object reference 146
programming model 61
resource strings 71
responding to user interaction 73
usage in Bridge 38
scrollbar objects 68, 158
sizing and placing windows and controls 166
slashes in pathnames 57
slider objects 67, 159
soliciting user input 38
special characters in pathnames 57
startup script locations 22, 238
statictext objects 66, 159
status line 30
strict ExtendScript directive 234
string translation
ExtendScript utilities 222
in ScriptUI windows 91

T

target ExtendScript directive 234
targets
message 237
script execution 234
selecting application for debugging 205
tasks, scheduling 53, 97
testing
in ExtendScript Toolkit 210
scripts 203
thumbnails
about 25
adding to Favorites 36, 111
deselecting 104
display style 101

- displaying web pages for 125
- events 109
- labels 126
- metadata 113
- navigating to 102
- object 123
- selecting 105
- sorting 102

Toolkit, ExtendScript 203
translation of UI strings 91, 222

U

unicode I/O 59
units of measurement 229
UnitValue object 229
URI notation 56
URLs, filtering and redirecting 103
user interaction
- Bridge events 106
- ScriptUI events and handlers 73

user interface controls 154
- accessing 64
- adding 63
- automatic layout 168
- creation properties 64
- dialogs for HTML 98
- grouping 62, 65
- methods 164
- placing 166
- properties 160
- removing 65
- responding to user interaction 165
- size and location 62
- types 65

user interface elements 61, 146
user interface options
- about 26, 38
- HTML dialogs 98
- navigation bars 116
- ScriptUI dialogs 40

user prompts 69, 148, 226
user-interaction events 27, 35

V

variable values during execution 206
VersionCue, loading local copies of referenced files 96
volumes, specifying in paths 58

W

web browser, embedded 42
web pages
- Content pane display properties 100, 125
- filtering and redirecting 103
- how they are displayed in Content pane 36, 123

Window class 147
windows
- accessing child controls 64
- adding controls 151
- automatic layout 75
- controlling 149
- creating 61, 148
- creation properties 64
- grouping controls 62, 150
- layout 62
- opening browsers 95
- placing 166
- removing child controls 65
- responding to user interaction 152
- reusing 147
- script access to 24, 28

X

XMP metadata 113