EBI Advanced CORBA Programming Workshop

IDLscript: Gluing CORBA Objects

Christophe Gransart

Université des Sciences et Technologies de Lille
Laboratoire d'Informatique Fondamentale de Lille
UPRES A 8022 CNRS
Bâtiment M3
59655 Villeneuve d'Ascq - France
http://corbaweb.lifl.fr

gransart@lifl.fr

Outline

- Why scripting Objects?
- A guided tour of IDLscript
- The IDLscript language
- The IDLscript binding
- Advanced Examples
- CorbaScript: the first implementation

Why scripting objects?

Main advantages of scripting objects

CORBA and Scripting

- OMG wants a scripting language for its component model (CORBA 3.0)
  - Making the composition of CORBA components easier
- Removing complexities such as
  - Memory allocation and de-allocation
  - Memory pointers
  - Compilation and linking procedures
- End-User Oriented
  - They only must focus on integration and business logic

General Scripting Benefits

- Interactive and interpreted: User-oriented
- To control and manage local resources
  - Files, processes, users: Unix shells, Perl
  - Graphical widgets: Tcl/Tk
  - Databases: SQL, OQL
- To assemble system components
  - Visual Basic, Emacs-Lisp, ...

General Scripting Benefits, cont'd

- Simplicity of use
  - Typeless, dynamic typing, garbage collector
- Easy to learn
  - Simpler than system programming languages
- Enhanced productivity
  - Easier development (interactive mode)
- Reduced costs
  - Reduce training and operating costs

CORBA and Scripting Language

- Scripting benefits can be applied to CORBA in all of the development and execution steps:
  - Design and prototyping
  - Development and testing
  - Configuration and administration
  - Discovering and using services
  - Assembling software components
  - Managing evolution

Some comments about the CORBA Scripting Language

- Requirements:
  - Must be object-oriented to reflect the CORBA object / component model
  - Must seamlessly integrate the OMG IDL type system
- Two approaches:
  - Take an existing scripting language and map OMG IDL concepts to its type system
  - Design a new scripting language based on the OMG IDL type system

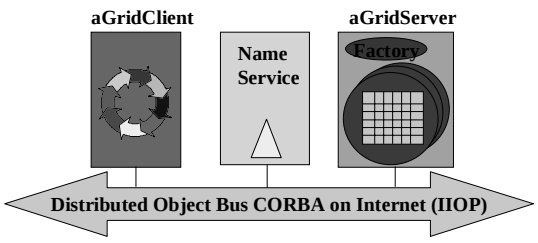
A guided tour of IDLscript

Presentation of an example and how to use IDLscript

IDLscript

- A new scripting language dedicated to CORBA
  - general purpose, interpreted and interactive
  - object-oriented and "everything is an object"
  - dynamic typing, reflexivity and introspection
- A complete and dynamic OMG IDL binding
  - access directly and naturally to any IDL specifications
    - loaded from the Interface Repository
    - no stubs/skeletons generation
  - simple CORBA object binding (IOR, INS URL)
  - dynamic invocation of CORBA objects (DII)
  - dynamic implementation of CORBA objects (DSI)

A Grid Application Example



Grid Service IDL Specification

```
module GridService{
        typedef double Value ;
        struct Coord {unsigned short x, y;};
        exception InvalidCoord {Coord pos;};
        interface Grid {
                readonly attribute Coord dimension ;
                void set (in Coord pos, in Value val) raises (InvalidCoord);
                Value get (in Coord pos) raises (InvalidCoord);
                void destroy () ;
        };
        interface Factory {
                Grid create_grid (in Coord dim, in Value init_value);
        };
};
```

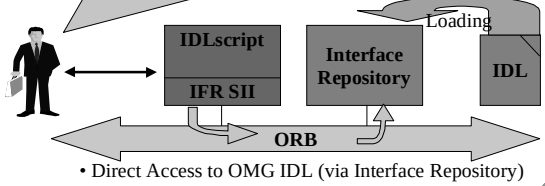
Some internal definitions

The Grid interface

The Factory interface

Discovering IDL Specifications

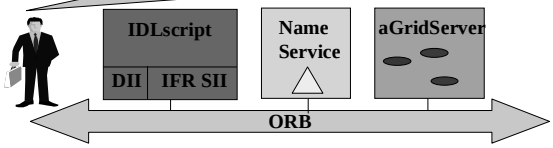
```
> GridService.Grid
<OMG-IDL interface GridService::Grid{…};>
> GirdService.Grid.set
<OMG-IDL void set (in Coord pos, in Value val)
        raises (InvalidCoord);>
```



- Direct Access to OMG IDL (via Interface Repository)
- End-users on-line help

Linking to Objects

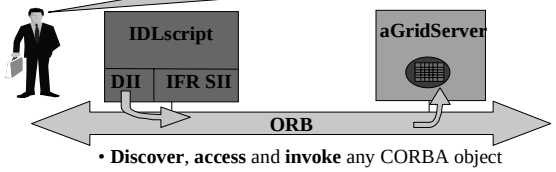
```
> NS = CORBA.ORB.resolve_initial _references(«NameService»)
> Gf = NS.resolve( factory_path )
> aGrid = Gf.create_grid ([20,5], 0)
```



- Initial object references
- Access to standard services: Naming, Trader, ...
- Object referencing with IOR and URL

Invoking Distributed Objects (1/2)

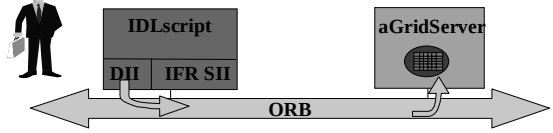
```
> aGrid.set ([1,2], 10)
> aGrid.dimension
GridService::Coord(20,5)
```



- **Discover**, **access** and **invoke** any CORBA object
- DII & IIOP = any operation on any object
- Dynamic type checking and conversions

Invoking Distributed Objects (2/2)

```
>  d = aGrid.dimension
>  for i in range (1, d.x) {
        for j in range (1, d.y) {
            print (aGrid.get ([i, j])) }}
0 10 0 0 ...
```

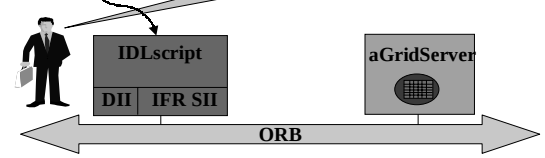


- Support complex users'activities

Procedures and Modules

GridTools.cs
proc displayGrid (g)
{ … }

```
> import GridTools
> GridTools.displayGrid (aGrid)
0 10 0 0 ...
```

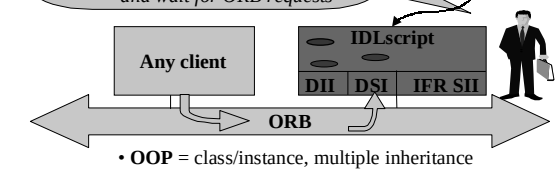


- Storing and reusing users'scripts

Implementing Distributed Objects

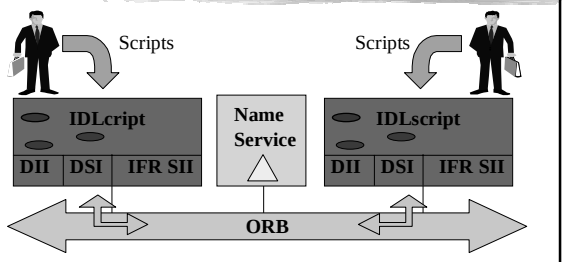
```
> import grid_impl
> factory = grid_impl.FACTORY ()
      register to a Name Service
      and wait for ORB requests
```

grid_impl.cs
class FACTORY{}
class GRID {}



- **OOP** = class/instance, multiple inheritance
- DSI = receive any CORBA request
- Create complex server applications

CORBA for End-Users



- Create complex client/server applications
- CORBA becomes simple and **user-friendly**

The IDLscript Language

Presentation of the language and its semantic

The IDLscript Language Core

- Lexical Rules
- Scripts
- Expressions
- Assignments
- Objects and Basic Types
- Control Flow Management
- Procedures
- Classes
- Exceptions
- Modules

Lexical Rules

- Comments begin by #
- Identifiers are case sensitive
- Few keywords (15)
- Some punctuation tokens
- Literals similar to OMG IDL
  - Integer
  - Floating-point
  - Character
  - String

Scripts

- A script is a set of statements
- A statement is
  - a null statement (;)
  - a statement block ({ statements })
  - an expression
  - a variable management statement
  - a control flow statement
  - a procedure declaration
  - a class declaration
  - an exception management statement
  - a module management statement

Expressions

- An expression is:
  - a parenthesed expression
  - a literal, an identifier, an array or dictionary creation
  - an arithmetic, relational, or logical expression
  - a procedural call
  - an attribute getting
  - a method call
  - an indexed getting
- The evaluation of an expression is an object
- The semantics depends on manipulated objects

Basic expressions

- Lexical literals:
  - `10`, `3.1415`, `'c'`, `«Hello World!»`
- Lexical identifiers refer to:
  - named objects : `Void`, `true`, `false`
  - objects contained into variables : `my_var`
- Array literals:
  - `[]`, `[ 10, 'c', «Hello World!», true]`
- Dictionary literals:
  - `{ «key»: «value», 2: true, 'c': 3.1415 }`

Expressions and operators

- Arithmetic operators:
  - unary ones: +, -
  - binary ones: +, -, *, /, %, \
- Relational operators:
  - ==, !=, <, <=, >, >=
- Logical operators:
  - negation: !
  - &&, ||
- Classical semantics for basic object types

Other expressions

- Procedural call
  - `expression(…)`
- Attribute getting
  - `expression.attribute` (or `expr!attr`)
- Method call
  - `expression.method(…)` (or `expr!method(…)`)
- Indexed getting
  - `expression[expression]`
- Note that semantics depends on `expression`
  - `A(10)` = procedure A calling or type A instanciation
  - `a!m(10)` = deferred call if a is a CORBA object ref.

Assignments

- Variable assignment
  - `my_var = expression`
  - declaration at the first assignment
- Attribute assignment
  - `object.attribute = expression` (or `o!a = e`)
  - semantics depends on the `object` behavior
- Indexed assignment
  - `object[expression] = expression`
  - semantics depends on the `object` behavior

Objects and Basic Types

- Every IDLscript entity is an object
- Every object has a type which defines
  - the `_type` attribute
  - the `_is_a(aType)` method
  - the `_toString()` method
- Every type is also an object
  - type conformity can be checked at run time
- Basic types are named by identifiers
  - `boolean`, `long`, `double`, `char`, `string`, `array`, and `dictionary`

Examples on Basic Types

```
>>> b = true
>>> b._toString()
«true»
>>> b._type
< type boolean { . . . } >
>>> b._type == boolean
true
>>> b._is_a(char)
false
>>> boolean._is_a(long)
false
```

- Ditto for other basic types (and on any types)

The `string` Type

- Strings are constant objects
- String objects support:
  - the `length` attribute `l = s.length`
  - the indexed getting notation `c = s[i]`
  - the + operator `s = s + c` `s = c + s`
  - all relational operators `b = s1 != s2`
  - various search methods: `index` and `rindex`
  - other methods: `substring`, `toLowerCase`, `toUpperCase`

The `array` Type

- An array object is a dynamic container of any IDLscript objects
- Array objects support:
  - the `length` attribute `l = a.length`
  - the indexed getting notation `o = a[i]`
  - the + operator `a = a1 + a2`
  - search methods: `contains`, `index` and `rindex`
  - other methods: `append`, `insert`, `delete`, `remove`
- The array type supports:
  - the `create` method `array.create(10)`

The `dictionary` type

- A dictionary object is a general container to store [key - value] associations
  - keys and values are of any types
- Dictionary objects support:
  - the `size` attribute `l = d.size`
  - the `keys` attribute `a = d.keys`
  - the `values` attribute `a = d.values`
  - the indexed getting notation `v = d[k]`
  - various methods: `contains`, `containsKey`, `remove`

Predefined Internal Procedures

- Evaluation of a stringified script
  - `eval(string)`
- Execution of a script file
  - `exec(string)`
- Read a string
  - `getline()`
- Print expressions
  - `print(…)`
  - `println(…)`

Five Control Flow Statements

- The `if` statement
  - `if (condition) { statements } else { statements }`
- The `while` statement
  - `while (condition) { statements }`
- The `do` statement
  - `do { statements } while(condition)`
- The `for` statement
  - `for variable in expression { statements }`
- The `return` statement
  - `return expression`

Procedures

- Declaration
  - `proc name(parameters) { statements }`
- Formal parameters are not statically typed
- Last parameters can have default values
- By default, variables are local
  - the `global` scope allows access to global variables
- Use of the `return` statement
- Procedures are objects
  - can be stored into variables, arrays, etc.

A Procedure Example

```
>>> proc up (a, b) { return a > b }
>>> proc down (a, b) { return a < b }
>>> proc sort (a, sort_criteria = up) {
      for i in range (0, a.length -2)
        for j in range (i + 1, a.length -1)
          if ( sort_criteria (a[i], a[j]) ) {
            temp = a[i] a[i] = a[j] a[j] = temp
          }
    }
>>> a = [ 60 , 6543 , 4 , 1124 , 1 ]
>>> sort (a)
>>> a
[1 , 4 , 60 , 124 , 6543]
>>> sort (a, down)
>>> a
[6543 , 124 , 60 , 4 , 1 ]
```

Classes

- Object-Oriented Programming:
  - Classes are user defined types
  - Polymorphism and overriding available
  - Multiple inheritance available
  - Overloading not provided
  - Class attributes and methods supported
  - Instance attributes and methods supported
- Declaration
  - **class** name(parent_classes) { statements }

Classes

- The `statements` part allows to define
  - Class attributes by assigning variables
  - Class methods by procedure declarations
  - Instance methods by procedure declarations
- Instance methods must have an explicit first parameter that will refer to the invoked instance
- A convention is used to name the initialization method called at the instanciation time
- Instance attributes are defined at their first assignment

A Simple Class Example

```
class Point2D {
  proc __Point2D__ (self,x,y) {
    self.x = x
    self.y = y
    Point2D.nb_created_points = Point2D.nb_created_points + 1
  }
  proc show (self) {
    println ("Point2D(x=", self.x, ", y=", self.y, ")")
  }
  proc move (self, x, y) {
    self.x = self.x + x
    self.y = self.y + y
  }
  proc how_many () {
    println (nb_created_points, " Point2D instances are been created.")
  }
  nb_created_points = 0
}
```

Using the Point2D Class

```
>>> p = Point2D(1,1)
>>> p
< Point2D instance
     x = 1
     y = 1
>
>>> p.move(10,10)
>>> p.show ()
Point2D(x=11, y=11)
>>> p._type
< class Point2D {
       proc __Point2D__ (self, x, y);
       proc show (self);
       proc move (self, x, y);
       proc how_many ();
       nb_created_points = 1;
} >
```

Class Inheritance Examples

```
class Point3D(Point2D) {
  proc __Point3D__ (self,x,y,z) {
    self.__Point2D__(x,y)
    self.z = z
  }
  move2D = Point2D.move
  proc move (self, p) {
    self.move2D (p.x, p.y)
    self.z = self.z + p.z
  }
}

class ColoredPoint2D (Point2D) { . . . }
class ColoredPoint3D (Point3D, ColoredPoint2D) { . . . }
```

Classes are types and objects

```
>>> t = ColoredPoint3D
>>> p = t(10,10,10,"green")
>>> p._type == ColoredPoint3D
true
>>> p._type == Point2D
false
>>> p._is_a(Point2D)
true
>>> ColoredPoint3D._is_a(Point2D)
true
```

Internal Exceptions

- An IDLscript interpreter must raise exceptions when execution problems occur:
  - BadArgumentNumber `a_string.toUpperCase(10)`
  - BadIndex `a_string[-1]`
  - BadTypeCoerce `a_string < 10`
  - ExecutionStopped `while(true) { . . .}`
  - NotFound `a_string.an_attribute`
  - NotSupported `s(10)`
  - Overflow `10 / 0`
  - ReadOnlyAttribute `s.length = 10`
  - SyntaxError `s.10`

Throwing User Exceptions

- Scripts can raise exceptions
  - **throw** expression
- Thrown exceptions are any IDLscript objects:
  - `throw 10`
  - `throw « A problem! »`
  - `throw [1, 2, 3]`
  - `throw Point2D(1,1)`

Handling Exceptions

- The `try/catch/finally` construct allows scripts to handle internal/user thrown exceptions
- Example:

```
try {
    . . . some script which throws an exception . . .
} catch (string e) {
    . . .
} catch (array e) {
    . . .
} catch (Point2D e) {
    . . .
} catch (e) {
    . . .
} finally {
    . . .
}
```

Modules

- Modules are loadable script files
  - text files with *.cs* extension
- Importation is done by:
  - **import** MyModule, AnotherModule
  - the *CSPATH* environment variable defines directories
- A module defines a scope containing
  - assigned variables `MyModule.a_var`
  - procedure declarations `MyModule.a_proc`
  - class declarations `MyModule.a_class`
- Modules are objects
  - can be stored into variables, arrays, etc.

The IDLscript Binding

Presentation of the binding of all OMG IDL constructions

How to create them and how to manipulate them

The OMG IDL Binding Principles

- IDLscript provides a dynamic OMG IDL binding
  - Access directly and naturally to any IDL specifications
  - Loaded from the Interface Repository
  - No stubs/skeletons generation
- The OMG IDL naming scheme is fully reflected:
  - IDL identifiers = IDLscript identifiers
  - IDL scopes = IDLscript scopes
  - `CORBA.Repository`
  - `CosNaming.NameComponent.id`

The OMG IDL Binding Principles

- The OMG IDL type system is fully reflected into the IDLscript type system:
  - basic types, modules, constants, enums, structs, unions, typedefs, sequences, arrays, exceptions, interfaces, TypeCodes, Anys, the CORBA module are reflected by IDLscript objects
  - excepts currently: value types and value boxes
- OMG IDL types are IDLscript types
  - `CosNaming.NamingContext.AlreadyBound`
  - instanciation by the IDLscript procedural call notation

OMG IDL Type Values

- OMG IDL type values are IDLscript objects
- Standard attributes and methods are
  - `_type`, `_is_a(aType)`, `_toString()`
- Specific attributes and methods are defined according to the OMG IDL type
  - *e.g.* IDL struct fields
- Semantic of expression notations is defined according to the OMG IDL type
  - *e.g.* the indexed getting notation for IDL sequences

Binding for Basic OMG IDL Types

- Basic OMG IDL types are reflected by IDLscript types contained into the CORBA scope
  - `CORBA.Void`, . . ., `CORBA.WString`
- Instanciation examples:
  - `CORBA.Short(1)`, `CORBA.ULong(10000)`
- Operators and automatic coercions between compatible IDL types and basic IDLscript types
  - `CORBA.Short(1) + 1 > CORBA.Double(3.1415)`

Binding for Basic OMG IDL Types

Basic OMG IDL Types	IDLscript Type Identifiers
void	**CORBA.Void**
short	**CORBA.Short**
unsigned short	**CORBA.UShort**
long	**CORBA.Long**
unsigned long	**CORBA.ULong**
long long	**CORBA.LongLong**
unsigned long long	**CORBA.ULongLong**
float	**CORBA.Float**

Binding for Basic OMG IDL Types

Basic OMG IDL Types	IDLscript Type Identifiers
double	**CORBA.Double**
long double	**CORBA.LongDouble**
boolean	**CORBA.Boolean**
char	**CORBA.Char**
wchar	**CORBA.WChar**
octet	**CORBA.Octet**
string	**CORBA.String**
wstring	**CORBA.WString**

Basic OMG IDL Values

- Examples:

```
>>> v1 = CORBA.Short(1)
>>> v2 = CORBA.ULong(10000)
>>> v1 + v2 > 100
true
>>> v3 = CORBA.String("Hello World!")
>>> v3.length
12
>>> v3 != ""
true
```

Binding for OMG IDL Module

- OMG IDL modules are IDLscript objects defining a scope to access to inner IDL definitions
- Inner definitions are accessed by the dotted notation
  - `GridService.Factory`
- Module evaluation displays its content
  - `>>> GridService`
    `< OMG-IDL module GridService { . . . }; >`

Binding for OMG IDL Module

- OMG IDL Examples

```
module GridService { ...};
module MA {
   module MB {...};
};
```

- IDLscript Representation

```
>>> GridService
< OMG-IDL module GridService { . . . }; >
>>> m = MA.MB
>>> m
< OMG-IDL module MA::MB { . . . }; >
```

Binding for OMG IDL Constant

- OMG IDL constants are IDLscript objects
  - can be prefixed by their module or interface scope
- Constant evaluation displays the IDL definition

```
>>> PI
< OMG-IDL const double PI = 3.14159; >
>>> Math.PI
< OMG-IDL const double Math::PI = 3.14159; >
```

Binding for OMG IDL Constant

- OMG IDL Examples

```
const double PI = 3.14159;
module Math {
   const double PI = 3.14159;
};
```

- IDLscript Representation

```
>>> PI
< OMG-IDL const double PI = 3.14159; >
>>> Math.PI
< OMG-IDL const double Math::PI = 3.14159; >
>>> c = PI
>>> c
< OMG-IDL const double PI = 3.14159; >
>>> c._type
< OMG-IDL typedef double CORBA.Double; >
```

Binding for OMG IDL Enum

- OMG IDL enumerations are IDLscript types defining scopes which contain enumeration items
- Enum evaluation displays the IDL definition

```
>>> Months
< OMG-IDL enum Months { January, February,
  March, April, May, June, July, August,
  September, October, November, December }; >
```

- The attribute getting notation is used for value creation

```
>>> Months.January
```

Binding for OMG IDL Enum

- OMG IDL Example

```
enum Months {
    January, February, March, April, May, June, July,
    August, September, October, November, December
};
```

- IDLscript Representation

```
>>> m = Months
>>> m
< OMG-IDL enum Months { January, February,
  March, April, May, June, July, August,
  September, October, November, December }; >
```

Binding for OMG IDL Enum

- Enum Values

```
>>> a = Months.January
>>> a
Months.January

>>> a._type
< OMG-IDL enum Months { January, February,
  March, April, May, June, July, August,
  September, October, November, December }; >

>>> a._is_a(Months)
true
```

Binding for OMG IDL Structure

- OMG IDL structs are IDLscript types
- Struct creation is done by the call notation
  - `StructIDLType(field1, …, fieldn)`
  - checks the number of fields
  - automatic coercion if needed
- IDL struct values are IDLscript objects
  - struct fields are reflected by IDLscript attributes
  - `v = a_struct.a_field`
  - `a_struct.a_field = v`
  - type checking when fields are set (+ automatic coercion)

Binding for OMG IDL Structure

- OMG IDL Examples

```
// This definition can be located inside or outside an IDL
   module or interface
struct Point {
    double x;
    double y;
};
struct TwoPoints {
    Point a;
    Point b;
};
```

Binding for OMG IDL Structure

- IDLscript Representation

```
>>> Point
< OMG-IDL struct Point { double x; double y; }; >

>>> Point.x
< OMG-IDL typedef double CORBA.Double; >

>>> TwoPoints
< OMG-IDL struct TwoPoints { Point a; Point b; }; >

>>> TwoPoints.a
< OMG-IDL struct Point { double x; double y; }; >

>>> a = Point
>>> a
< OMG-IDL struct Point { double x; double y; }; >
```

Binding for OMG IDL Structure

- Structure Value

```
>>> p1 = Point (1,2)
>>> p1
Point(1,2)
>>> tp1 = TwoPoints([11,22],[33,44])
>>> tp1
TwoPoints(Point(11,22),Point(33,44))

>>> tp2 = TwoPoints(p1,Point(3,4))

>>> tp3 = TwoPoints(Point(6,7),Point(8,9))
```

Binding for OMG IDL Structure

- Structure Fields

```
>>> p1.x
CORBA.Double(1)
>>> p1.x = -1
>>> p1
Point(-1,2)

>>> tp1.a
Point(11,22)
>>> tp1.a.y
CORBA.Double(22)

>>> tp1._type
< OMG-IDL struct TwoPoints { Point a; Point b; } >
```

Binding for OMG IDL Union

- OMG IDL unions are IDLscript types
- Union creation is done by the call notation
  - `UnionIDLType(discriminator,value)`
  - checks the discriminator and the value
  - automatic coercion if needed
- IDL union values are IDLscript objects
  - discriminator is reflected by the _d readonly attribute
  - union fields are reflected by IDLscript attributes
  - union field getting is checked according to the discr.
  - discriminator is set at field assignment

Binding for OMG IDL Union

- OMG IDL Examples

```
// This definition can be located inside or outside an IDL
   module or interface
union Union switch(unsigned short) {
    case 0: short m_short;
    case 1: long m_long;
    case 2: float m_float;
};
```

Binding for OMG IDL Union

- IDLscript Representation

```
>>> u = Union
>>> u
< OMG-IDL union Union switch (unsigned short)
  {
        case 0: short m_short;
        case 1: long m_long;
        case 2: float m_float;
}; >

>>> u == Union
true
```

Binding for OMG IDL Union

- Union Values

```
>>> a = Union(0,1)
>>> a
Union(0,1)

>>> b = Union(2,10.3)
>>> b
Union(2,10.3)

>>> a._type == b._type
true
```

Binding for OMG IDL Union

- Union Fields

```
>>> a._d
CORBA.UShort(0)

>>> a.m_short
CORBA.Short(1)

>>> a.m_long = 2
>>> a.m_long
CORBA.Long(2)
>>> a._d
CORBA.UShort(1)
```

Binding for OMG IDL Typedef

- OMG IDL typedefs are IDLscript types
  - the typedef type is conform to its aliased type
- Typedef creation is done by the call notation
  - `TypedefIDLType(field1, …, fieldn)`
  - checks the number of fields according to the aliased type
  - automatic coercion if needed
- Typedef values are used according to the rules defined for the binding of the aliased type

Binding for OMG IDL Typedef

- OMG IDL Examples

```
// This definition can be located inside or outside an IDL
   module or interface
typedef unsigned short Day;
typedef Point Coordinate;
```

- IDLscript Representation

```
>>> Day
< OMG-IDL typedef unsigned short Day; >

>>> c = Coordinate
>>> c
< OMG-IDL typedef Point Coordinate; >

>>> c.x
< OMG-IDL typedef double CORBA.Double; >
```

Binding for OMG IDL Typedef

- Typedef Values

```
>>> d = Day(2)
>>> d
Day(2)

>>> c = Coordinate(1.1,2.2)
>>> c
Coordinate(1.1,2.2)

>>> c.x
CORBA.Double(1.1)

>>> c._is_a(Point)
true
```

Binding for OMG IDL Sequence

- OMG IDL sequences are IDLscript types
  - anonymous sequence not allowed
- Sequence creation is done by the call notation
  - `SequenceIDLType(item1, …, itemn)`
  - type checking of items according to the seq. item type
  - automatic coercion if needed
- IDL sequence values are IDLscript objects
  - with the `length` attribute
  - items are accessed by the indexed notation
    - `v = a_sequence[index]`
    - `a_sequence[index] = v`
  - type checking when items are set (+ automatic coercion)

Binding for OMG IDL Sequence

- OMG IDL Examples

```
// This definition can be located inside or outside an IDL
   module or interface
typedef sequence<string> SeqString;
typedef sequence<Months> SeqMonths;
typedef sequence<Point> SeqPoint;
```

Binding for OMG IDL Sequence

- IDLscript Representation

```
>>> SeqString
< OMG-IDL typedef sequence<string> SeqString; >

>>> SeqMonths
< OMG-IDL typedef sequence<Months> SeqMonths; >

>>> s = SeqPoint
>>> s
< OMG-IDL typedef sequence<Point> SeqPoint; >
```

Binding for OMG IDL Sequence

- Sequence Values

```
>>> s = SeqString("One","Two","Three")
>>> s
SeqString("One","Two","Three")

>>> s = SeqMonths()
>>> s
SeqMonths()

>>> s = SeqPoint ( [1.1,2.2] ,[3.3,4.4] ,[5.5,6.6] )
>>> s
SeqPoint(Point(1.1,2.2),Point(3.3,4.4),Point(5.5,6.6))

>>> s1 = SeqPoint ( [1.1,2.2], Point(3.3,4.4),
  Point(CORBA.Double(5.5), CORBA.Double(6.6)) )
>>> s1._type
< OMG-IDL typedef sequence<Point> SeqPoint; >
```

Binding for OMG IDL Sequence

- Sequence Items

```
>>> s1[0]
Point(1.1,2.2)

>>> s1[0] = [100,200]

>>> s1[1].x = 300

>>> s1.length
3

>>> for i in s1 { println (i) }
Point(100,200)
Point(300,4.4)
Point(5.5,6.6)
```

Binding for OMG IDL Array

- OMG IDL arrays are IDLscript types
  - anonymous array not allowed
- Array creation is done by the call notation
  - `ArrayIDLType(item1, …, itemn)`
  - number of items must be equal to the array size
- IDL array values are IDLscript objects
  - with the `length` attribute
  - items are accessed by the indexed notation
    - `v = an_array[index]`
    - `an_array[index] = v`
- Item type checking according to the array item type and automatic coercion if needed

Binding for OMG IDL Array

- OMG IDL Examples

```
// This definition can be located inside or outside an IDL
   module or interface
typedef long ArrayLong[10];
typedef Point ArrayPoint[10];
```

- IDLscript Representation

```
>>> ArrayLong
< OMG-IDL typedef long[10] ArrayLong;>

>>> a = ArrayPoint
>>> a
< OMG-IDL typedef Point[10] ArrayPoint;>
```

Binding for OMG IDL Array

- Array Values

```
>>> a = ArrayLong(1,2,3,4,5)
Exception : < BadArraySize: array must have 10 items >
File "stdin", line 1 in ?

>>> a = ArrayLong(1,2,3,4,5,6,7,8,9,10)
>>> a
ArrayLong(1,2,3,4,5,6,7,8,9,10)

>>> a = ArrayPoint([1,1],[2,2],[3,3],[4,4],[5,5],[6,6]
[7,7],[8,8],[9,9],[10,10])
>>> a
ArrayPoint(Point(1,1),Point(2,2),Point(3,3),Point(4,4)
Point(5,5),Point(6,6),Point(7,7),Point(8,8),Point(9,9)
Point(10,1 0))

>>> a._type == ArrayPoint
true
```

Binding for OMG IDL Array

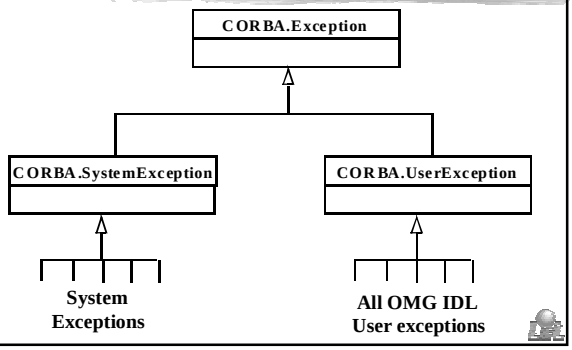
- Array Items

```
>>> a[0]
Point(1,1)

>>> a[0] = [100,100]

>>> a[1].x = 200

>>> a.length
10

>>> for i in a { println (i) }
Point(100,100)
Point(200,2)
Point(3,3)
...
```

Binding for OMG IDL Exceptions

- OMG IDL exceptions are IDLscript types
- IDLscript provides a hierarchical type graph containing all CORBA exceptions
- `CORBA.Exception`
  - the root of the hierarchy of CORBA exceptions
- `CORBA.SystemException`
  - the super type of any system exceptions
- `CORBA.UserException`
  - the super type of any user defined exceptions

CORBA Exceptions Type Graph



Handling CORBA Exception

- Handling CORBA exceptions with the IDLscript `throw/try/catch/finally` mechanism

```
try {
    # a script code.
    throw CORBA.UNKNOWN()
} catch (CosNaming.NamingContext.AlreadyBound ae) {
    . . .
} catch (CORBA.UserException ue) {
    . . .
} catch (CORBA.SystemException se) {
    . . .
} finally {
    # a finally script code.
}
```

CORBA System Exceptions

- System exceptions are IDLscript types
  - subtypes of the `CORBA.SystemException`
  - defined into the CORBA scope
    - `CORBA.UNKNOWN, . . ., CORBA.INVALID_TRANSACTION`
    - also the `CORBA.CompletionStatus` enumeration
- Creation is done by the procedural call notation
  - `SystemExceptionIDLType(minor,completed)`
  - `minor` and `completed` can be omitted
- System exception values are IDLscript objects
  - with `minor` and `completed` readonly attributes

All CORBA System Exceptions

```
CORBA.UNKNOWN                  CORBA.BAD_PARAM
CORBA.NO_MEMORY                CORBA.IMP_LIMIT
CORBA.COMM_FAILURE             CORBA.INV_OBJREF
CORBA.NO_PERMISSION            CORBA.INTERNAL
CORBA.MARSHAL                  CORBA.INITIALIZE
CORBA.NO_IMPLEMENT             CORBA.BAD_TYPECODE
CORBA.BAD_OPERATION            CORBA.NO_RESOURCES
CORBA.NO_RESPONSE              CORBA.PERSIST_STORE
CORBA.BAD_INV_ORDER            CORBA.TRANSIENT
CORBA.FREE_MEM                 CORBA.INV_IDENT
CORBA.INV_FLAG                 CORBA.BAD_CONTEXT
CORBA.OBJ_ADAPTER              CORBA.DATA_CONVERSION
CORBA.OBJECT_NOT_EXIST         CORBA.INTF_REPOS
CORBA.TRANSACTION_REQUIRED     CORBA.TRANSACTION_ROLLEDBACK
CORBA.INVALID_TRANSACTION
```

Binding for OMG IDL Exception

- System Exception Type

```
>>> CORBA.UNKNOWN
< OMG-IDL exception CORBA::UNKNOWN {
        unsigned long minor;
        CORBA::CompletionStatus completed;
}; >

>>> CORBA.CompletionStatus
< OMG-IDL enum CORBA::CompletionStatus {
 COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE
}; >

>>> CORBA.UNKNOWN._is_a(CORBA.Exception)
true
```

Binding for OMG IDL Exception

- System Exception Type

```
>>> e = CORBA.UNKNOWN
>>> e._is_a(CORBA.SystemException)
true

>>> e._is_a(CORBA.UserException)
false
```

Binding for OMG IDL Exception

- System Exception Value

```
>>> s = CORBA.UNKNOWN()
>>> s = CORBA.UNKNOWN(100)
>>> s = CORBA.UNKNOWN(100,
        CORBA.CompletionStatus.COMPLETED_YES)

>>> s.minor
100
>>> s.completed
CORBA.CompletionStatus.COMPLETED_YES
>>> s._type == CORBA.UNKNOWN
true
```

Binding for OMG IDL Exception

- System Exception Value

```
>>> s._is_a (CORBA.Exception)
true

>>> s._is_a (CORBA.SystemException)
true

>>> s._is_a (CORBA.UserException)
false
```

Binding for OMG IDL Exception

- User exceptions are IDLscript types
  - subtypes of the `CORBA.UserException`
- Exception creation is done by the call notation
  - `UserExceptionIDLType(field1, …, fieldn)`
  - checks the number of fields
  - automatic coercion if needed
- IDL exception values are IDLscript objects
  - exception fields are IDLscript attributes
    - `v = an_exception.a_field`
    - `an_exception.a_field = v`
  - type checking when fields are set (+ automatic coercion)

Binding for OMG IDL Exception

- User Exception OMG Example

```
// This definition can be located inside or outside an IDL
   module or interface
exception EmptyException {};

exception Exception {
    string s;
    Months m;
    Point p;
};
```

Binding for OMG IDL Exception

- IDLscript Representation

```
>>> EmptyException
< OMG-IDL exception EmptyException {}; >

>>> Exception
< OMG-IDL exception Exception {
    string s;
    Months m;
    Point p;
}; >

>>> Exception.p
< OMG-IDL struct Point {
    double x;
    double y;
}; >
```

Binding for OMG IDL Exception

- IDLscript Representation

```
>>> Exception._is_a(CORBA.Exception)
true

>>> e = Exception
>>> e._is_a(CORBA.SystemException)
false

>>> e._is_a(CORBA.UserException)
true
```

Binding for OMG IDL Exception

- User Exception Value

```
>>> u = EmptyException()
>>> u = Exception ("Hello", Months.June,
  [100,100])
>>> u
Exception("Hello",Months.June,Point(100,100))

>>> u.s
"Hello"

>>> u._is_a (CORBA.Exception)
true
>>> u._is_a (CORBA.SystemException)
false
>>> u._is_a (CORBA.UserException)
true
```

Binding for OMG IDL Interface

- OMG IDL interfaces are IDLscript types
- IDL inheritance graph is reflected by the same IDLscript type graph
- CORBA object references are IDLscript objects with attributes and operations according to their OMG IDL interface
- Invocations are done through the DII
  - parameter type checking is done via the IFR
  - all parameter passing modes are available
  - twoways, oneway and deferred modes are supported

Binding for OMG IDL Interface

- OMG IDL Example:

```
interface Foo {
    attribute string assignable;
    readonly attribute double nonassignable;
    long method(in long p1) raises(EmptyException);
};

interface AnotherFoo : Foo {
    long operation(in long p1, out long p2, inout long p3)
            raises(EmptyException);
};
```

Binding for OMG IDL Interface

- IDLscript Representation

```
>>> Foo
< OMG-IDL interface Foo {
    attribute string assignable;
    attribute readonly double nonassignable;
    long method (in long p1)
  raises(EmptyException);
} >

>>> a = AnotherFoo
>>> a
< OMG-IDL interface AnotherFoo : Foo {
    long operation (in long p1, out long p2,
  inout long p3) raises(EmptyException);
} >
```

Binding for OMG IDL Interface

- IDLscript Representation

```
>>> a = AnotherFoo.assignable
>>> a
< OMG-IDL attribute string Foo::assignable >

>>> AnotherFoo.operation
< OMG-IDL operation long AnotherFoo::operation
  (in long p1, out long p2,   inout long p3)
  raises(EmptyException) >

>>> AnotherFoo._is_a (Foo)
true
```

Binding to CORBA Objects

- Binding to CORBA objects:
  - `InterfaceIDLName (StringifiedObjectReference)`
- Stringified object references:
  - standard IOR format
  - standard INS URL formats
- Examples:

```
objref = Foo ("IOR:...")
objref = Foo ("iioploc://www.lifl.fr:10000/my_foo")
# Check if the IOR/URL refers to a Foo object (~ narrow)
objref = CORBA.Object ("IOR:...")
# Automatic narrowing to the most derivated IDL interface
```

Binding for OMG IDL Interface

- Object References

```
>>> objref = CORBA.Object("IOR:.....")
>>> objref._type
< OMG-IDL interface AnotherFoo : Foo {
      long operation (in long p1, out long p2,
  inout long p3)
  raises(EmptyException);
} >

>>> objref = AnotherFoo("IOR:.....")
>>> objref =
  AnotherFoo("iioploc://host:port/name")

>>> objref._is_a(Foo)
true
```

Access to OMG IDL Attributes

- Simple and direct access
- Automatic conversions from/to IDL values

```
>>> objref.assignable = "Hello world"
>>> println (objref.assignable, '!')
Hello world!
>>> objref.nonassignable = 10
Exception: < ReadOnlyAttribute: ... >
```

- Transparently done through the DII
- Control, coercion done by IFR information

Binding for OMG IDL Interface

- Access to OMG IDL Attributes

```
>>> objref.assignable = "Hello World"
>>> println(objref.assignable, '!')
Hello World!


>>> objref.nonassignable = 10
Exception: < ReadOnlyAttribute: < attribute
  readonly double Foo::nonassignable; > >
      File "stdin", line 2 in ?
```

Invocation of OMG IDL Operations

- Invocations are done via the DII
  - `objref.method (p1, …, pn)`
- Parameters control and coercion via the IFR
- Exception management
- Examples:

```
>>> objref.method(100)
100
>>> try {
      r = objref.method(100)
    } catch (EmptyException e) {
      . . .
    }
```

Parameter Passing Modes

- *in* parameters are transmitted by values
- *inout* and *out* require to use an `Holder` object
- Example:

```
>>> out = Holder ()
>>> inout = Holder (200)
>>> objref.operation (100, out, inout)
100
>>> out.value
300
```

- An `Holder` contains an attribute: **value**

Invocation of Oneway Operations

- Oneway operations are invoked in the same way that twoways operations
  - `objref.operation(…)`
- They are always called asynchronously via the DII

Deferred Operation Invocations

- Deferred invocations of OMG IDL operations are supported by IDLscript
  - the method calling notation is `object!method(…)`
- The result is a `FutureReply` object with
  - the `value` attribute : the result (wait if needed)
  - the `poll()` method : polls if the request is completed
  - the `wait()` method : waits for the reply
- `out/inout` parameters are `Holder` objects which contain `FutureReply` objects

Binding for OMG IDL Interface

- Operation Invocation Using the Deferred Mode

```
>>> objref.method
< OMG-IDL operation long Foo::method (in long
  p1) raises(EmptyException) >

>>> futureReply = objref!method(100)
...
>>> futureReply.value
100
```

The `FutureReply` Type

Functionality	Explanation
futureReply.value	waits for the end of the invocation and returns the result.
futureReply.poll()	polls the end of the invocation and returns a boolean: true = invocation is completed ; false = invocation is still running.
futureReply.wait()	waits for the end of the invocation.

Binding for OMG IDL Interface

- Using *inout* and *out* Parameters

```
>>> out = Holder()
>>> inout = Holder(200)
>>> futureReply = objref!operation (100, out, inout)
...
>>> futureReply.value
100
>>> myFutureReplyForMyOutParameter = out.value
>>> myFutureReplyForMyOutParameter.value
300
```

Implementing OMG IDL Interfaces

- Done by IDLscript classes
- Oper./attr. implemented by instance methods
  - IDL attributes
    - proc **_get_**<attr_name>(self) { return value }
    - proc **_set_**<attr_name>(self,value) { . . . }
  - IDL operations
    - an instance method with the same name
    - proc <operation_name> (self, . . .) { . . . }
- Incoming requests are managed through the DSI

Implementation of the Interface Foo

```
class FOO {
  proc __FOO__ (self, s, d) { self.s = s
                              self.d = d }
  proc _get_assignable (self) { return self.s }
  proc _set_assignable (self, value) { self.s = value }
  proc _get_nonassignable (self) { return self.d }
  proc method (self, p1) {
    if ( p1 == 0 ) { throw EmptyException() }
    return p1
  }
}
```

Implementation of the Interface AnotherFoo

```
class AnotherFOO (FOO) {
  proc __AnotherFOO__ (self, s, d)  {
                       self.__FOO__(s,d) }
  proc operation (self, p1, p2, p3) {
    if ( p1 == 0 ) { throw EmptyException() }
    p2.value = p1 + p3.value
    return p1
  }
}
```

Object Registration

- Provided by two approaches
- The POA approach:
  - `PortableServer::Servant` is IDLscript class instance
- A Java-like approach:
  - `CORBA.ORB.connect()` and `disconnect()` methods
  - Connections may be done explicitly or implicitly
  - Disconnections are always done explicitly

Object Registration Example

```
>>> a_foo = FOO ("Hello",10)
>>> # 'a_foo' refers to a FOO instance.
>>> CORBA.ORB.connect(a_foo, Foo, "my_foo")
>>> # 'a_foo' is now associated to a Foo CORBA object.
>>> # The 'a_foo' instance becomes accessible from
>>> # the ORB. The last parameter is optional.
>>> a_foo._this
< DSI Object Foo("IOR: . . .") >
>>> # The '_this' attribute refers to the associated
>>> # DSI object.
>>> # This is the CORBA object reference implemented
>>> # by the 'a_foo' instance.
>>> ...
>>> CORBA.ORB.disconnect(a_foo)
>>> # Explicit disconnection. The 'a_foo'
>>> # instance becomes inaccessible from the ORB.
```

Object Adapter Run-Time Exceptions

- `CORBA::BAD_OPERATION`
  - the invoked IDL operation is not supported by the implementation
- `CORBA::OBJ_ADAPTER`
  - the servant object has been explicitly disconnected
- `CORBA::NO_IMPLEMENT`
  - the servant object does not provide implementation
- `CORBA::BAD_INV_ORDER`
  - internal exception during operation execution

Binding for OMG IDL TypeCode

- The IDL TypeCode type is reflected by the IDLscript `CORBA.TypeCode` type
  - with standard operations: `equal`, `kind`, ...
  - also the `CORBA.TCKind` enumeration
- All IDL type reflections are `CORBA.TypeCode` values

```
// OMG IDL
interface ExampleTC { void send (in TypeCode tc); };
>>> o = ExampleTC("IOR:. . . ")
>>> o.send(CORBA.Long)
>>> o.send(Point)
>>> o.send(Foo)
```

Binding for OMG IDL TypeCode

- OMG IDL Example

```
interface ExampleTC {
    void send (in TypeCode tc);
};
```

- IDLscript Representation

```
>>> o = ExampleTC("IOR:....")
>>> o.send(CORBA.Long)
>>> o.send(Point)
>>> o.send(Foo)
>>> tc = CORBA.TypeCode(Foo)
>>> o.send(tc)
```

The CORBA.TypeCode Type

tc.equal(aCorbaType)	tc.member_type(anIndex)
tc.kind()	tc.member_label(anIndex)
tc.id()	tc.discriminator_type()
tc.name()	tc.default_index()
tc.member_count()	tc.length()
tc.member_name(anIndex)	tc.content_type()

Binding for OMG IDL Any

- The IDL Any type is reflected by the IDLscript CORBA.Any type
  - with two attributes: value and type
- The reflection of any IDL type and value can be coerced to a CORBA.Any value

```
// OMG IDL
interface ExampleAny {  void send (in any a); };
>>> o = ExampleAny("IOR:....")
>>> o.send(CORBA.Long(10))
>>> o.send(Point(10,10))
>>> o.send(Foo("IOR:...."))
>>> o.send(AnotherFoo)
```

Binding for OMG IDL Any

- OMG IDL Example

```
interface ExampleAny {
    void send (in any a);
};
```

- IDLscript Representation

```
>>> p = Point(10,10)
>>> foo = Foo("IOR:....")
>>> o = ExampleAny("IOR:....")
>>> o.send(CORBA.Long(10))
>>> o.send(p)
>>> o.send(foo)
>>> o.send(AnotherFoo)
```

Binding for OMG IDL Any

- IDLscript Representation

```
>>> a = CORBA.Any(p)
>>> a
CORBA.Any(Point(10,10))
>>> o.send(a)

>>> a.type
< OMG-IDL struct Point {
    double x;
    double y;
}; >
>>> a.value
Point(10,10)
```

Binding for OMG IDL Any

- CORBA.Any Implicit Conversions

Type	Conversion to
a long L	CORBA::Any(CORBA::Long(L))
a double D	CORBA::Any(CORBA::Double(D))
a char C	CORBA::Any(CORBA::Char(C))
a boolean B	CORBA::Any(CORBA::Boolean(B))
a string S	CORBA::Any(CORBA::String(S))

Binding for OMG IDL Value

- OMG IDL value types are IDLscript types
  - reflection of the IDL inheritance graph
- Creation of IDL boxed value is achieved by the calling notation according to the boxed type
  - same number of arguments, type conformity
- Creation of concrete value is done by calling an initializer declared in the value type
  - argument number and types must be conform to the initializer signature
  - abstract value types can not be instanciated
- Null value = `ValueType._null`

Value Manipulation

- Boxed values are manipulated like objects of the boxed type
- Concrete and abstract values are manipulated using the dotted notation
- Concrete value operations can be invoked if the value is associated with a local implementation
- Public state members can be get and set using the dotted notation
- Private state members are not accessible

Implementing OMG IDL Values

- Done by IDLscript classes
- Public and private IDL value members represented by instance attributes
  - same name as in OMG IDL definition
  - explicitly assigned by initializers
  - `CORBA::MARSHAL` thrown if not assigned
- Initializers and operations implemented by instance methods
  - same name as in OMG IDL definition
  - `CORBA::NO_IMPLEMENT` thrown if called but not implemented

Factory Registration

- `CORBA::ValueFactory` native type is reflected by IDLscript classes
- Classes must be explicitly registered
  - `OMG IDL: valuetype Employee { . . . };`

```
>>> class EMPLOYEE { . . . }

>>> orb.register_value_factory(
                "IDL:Employee:1.0 ", EMPLOYEE)

>>> orb.register_value_factory(
                Employee.id(), EMPLOYEE)
```

Custom Values

- Implementation classes of custom marshaled values must implement marshaling and unmarshaling methods

```
class IMPL {
    proc marshal (self, dos) {
        dos.write_boolean (self.state1)
        . . .
    }
    proc unmarshal (self, dis) {
        self.state1 = dis.read_boolean ()
        . . .
    }
}
```

- Reflection of `CORBA::DataOutputStream` and `CORBA::DataInputStream` value types

The Global CORBA Object

- Reflection of the CORBA module
- Contains the hierarchy of objects
  - basic OMG IDL types (`Long`, `Double`, `String`, ...)
  - `TypeCode` and `Any` types
  - some basic enums (`TCKind`, `CompletionStatus`)
  - standard exception types
  - the `Object` type with its standard operations
  - the `ORB` singleton object with its standard operations
    - `+ connect, disconnect`

The `CORBA::Object` Object

- Reflection of `CORBA::Object` interface
- Standard operations prefixed by ' _ '
  - `_get_interface`, `_hash`, `_is_equivalent`, ...
- `duplicate` and `release` are not reflected
  - IDLscript includes a garbage collector
- `_is_a` operation uses a IDLscript type
- `create_request` is not reflected
  - IDLscript offers a user-friendly calling notation
  - the engine must use the DII

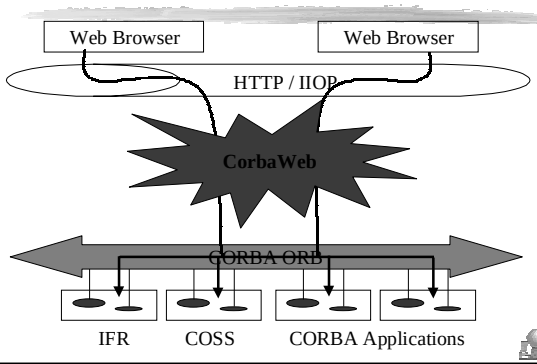
The `CORBA::ORB` Object

- Reflection of ORB singleton object
- Provides standard operations
  - `object_to_string`, `list_initial_references`, `run`, ...
- DII related operations are not reflected
  - IDLscript offers a user-friendly calling notation
- TypeCode creation is not reflected
  - all OMG IDL definitions are automatically available (IFR)
- For value factory related operations
  - `CORBA::ValueFactory` = IDLscript class

Advanced Examples

CorbaWeb
Remote Scripting Engine

The CorbaWeb architecture



Some Technical Points about CorbaWeb

- 1 HTTP server + 1 IDLscript interpreter
  - A CGI program
  - An HTTP micro-server
  - An Apache module
- URLs for IDLcript scripts
  - interpreted on the fly
  - to generate WWW documents
- URLs to specify HTML documents + IDLscript codes
  - HTML documents dynamic generation

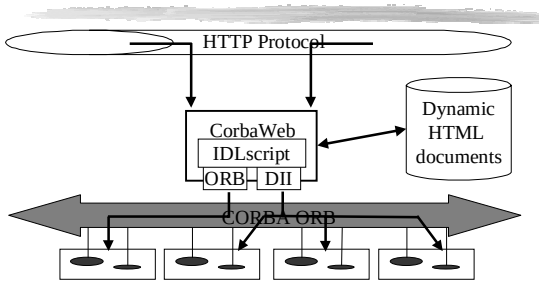
The CorbaWeb Functionalities

- **Object (one or more) representation**
  - on the fly generation of MIME documents
- **Browsing through object graphs**
  - on the fly generation of HTML anchors
  - URL contains IOR of the visited object
- **Operation invocations on objects**
  - on the fly HTML form generation with information from the Interface Repository
  - operation invocations and script mobile execution invocations
    scripts and IORs are carried by HTTP

OMG IDL / HTML Mapping

OMG IDL	HTML
Simple value (integer, string, boolean,enum, ...)	HTML text
Array, sequence and structure	Recursively, each element is mapped to HTML
Object reference	An HTML anchor refers to the representation script for the object
Interface	A set of HTML forms (one for each operation of attribute)
Attribut	One form to read the value One to set the value (if R/W)
Operation and parameters	One form to invoke the operation a set of widgets (menu, scrolling list, check box, text field)

CorbaWeb: Dynamic Documents



- On the fly documents generation (using patterns)

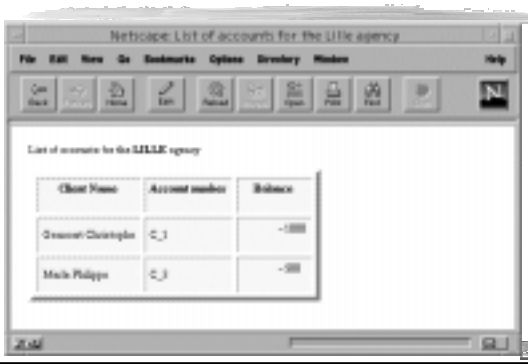
HTML + IDLscript Example

```
[ # Initializing IDLscript code
  NS = CORBA.ORB.resolve_initial_references("NameService")
  agency = NS.resolve ( [ ["bank",""], ["agency",""] ] )
  town = agency.address.town
]
<HTML> <HEAD>
<TITLE>List of accounts of the
[ # display the name of the town
      print (town)
] agency
</TITLE> </HEAD> <BODY>
<TABLE ALIGN=ABSCENTER BORDER=5 CELLSPACING=5 CELLPADDING=5>
<CAPTION>List of accounts of the
 [ print (town) ] agency
</CAPTION>
```

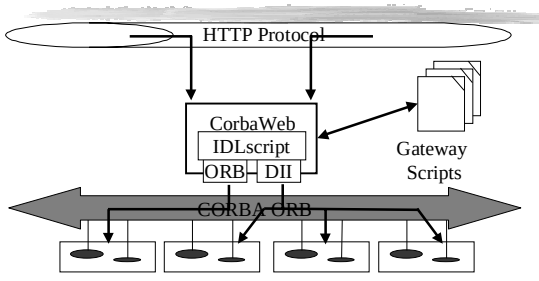
HTML + IDLscript Example

```
<TR> <TH>Client Name<TH>Account number
<TH>Balance </TR>
[ # display the « bad » accounts
  clients = agency.client_list ()
  for client in clients {
    client_name = client.name
    accounts = client.accounts_list ()
    for account in accounts {
      balance = account.balance
      if ( balance < 0 ) {
        print ("<TR>")
        print ("<TD> ", client_name, "</TD>")
        print ("<TD> ", account.number, "</TD>")
        print ("<TD ALIGN=RIGHT> ", balance, "</TD>")
        println ("</TR>")
} } }
]
</TABLE> </BODY> </HTML>
```

Generated Document



CorbaWeb: Gateway Scripts



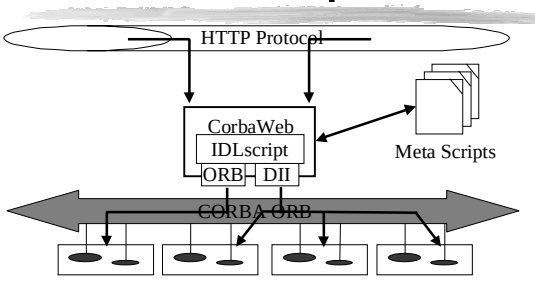
- Dedicated access to one or several CORBA objects
- Operation representations and invocations

CorbaWeb: Gateway Script for the Grid Example

```
module GridService{
        struct Coord {unsigned short x, y;};
        exception InvalidCoord {Coord pos;};
        typedef double Value ;
        interface Grid {
                readonly attribute Coord dimension ;
                void set (in Coord pos, in Value val) raise (InvalidCoord);
                Value get (in Coord pos) raises (InvalidCoord);
                void destroy () ;
        };
        interface Factory {
                Grid create_grid (in Coord dim, in Value init_value);
        };
};
```

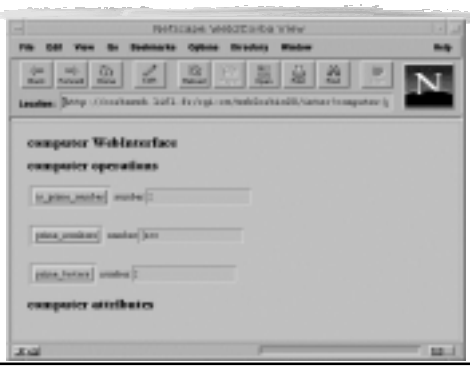
CorbaWeb: a Gateway Script for the Grid Example

CorbaWeb: Meta Scripts

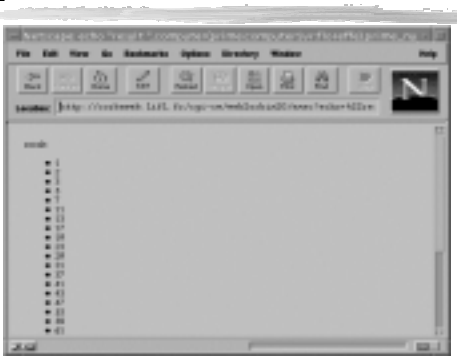


- Generic actions on all kinds of CORBA objects

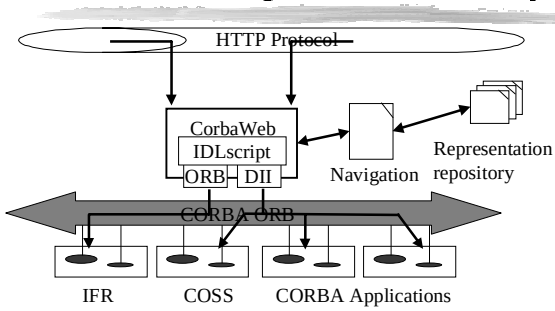
GUI Generation for an OMG IDL Interface



Operation Invocation

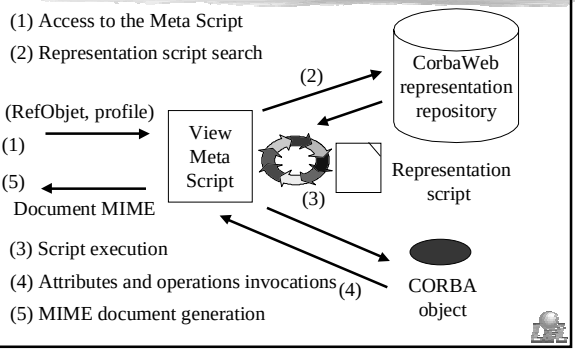


CorbaWeb: Navigation Meta Script

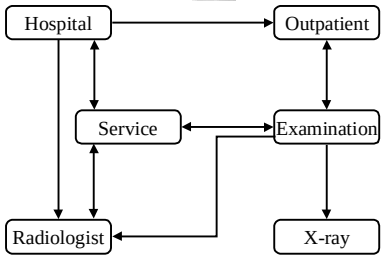


- Object browsing personalization using a representation repository

Navigation Meta Script



An Information Sytem from an Hospital



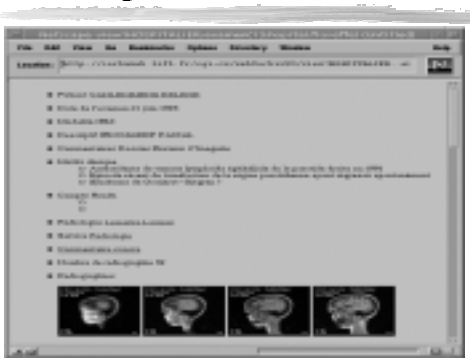
- Described by OMG IDL interfaces
- Implemented with C++

Browsing the Hospital Object

Browsing the Service Object

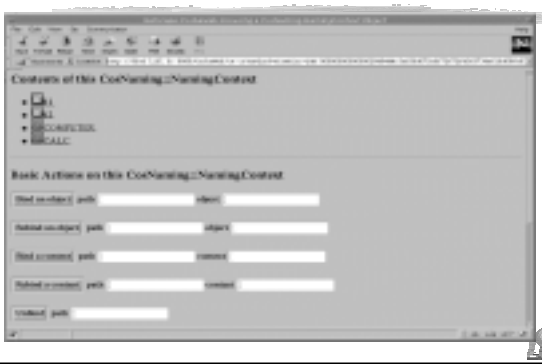
Browsing the Radiologist Object

Browsing an Examination

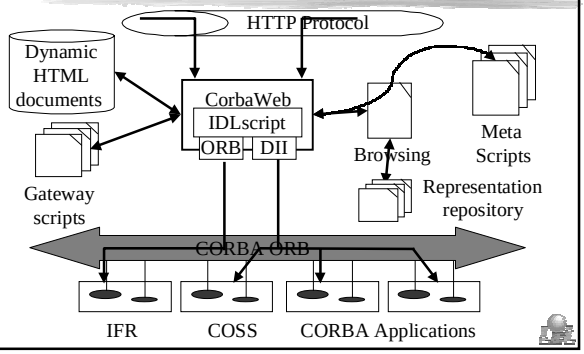


Browsing an X-ray

CorbaWeb: Name Service Administration

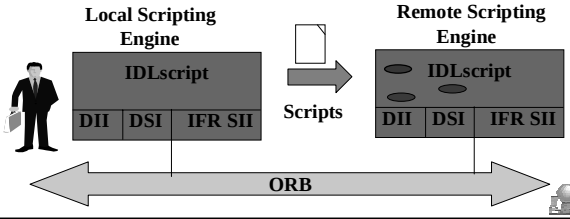


CorbaWeb: an Extensible Frawemork



Remote Scripting Engine

- A CORBA application to execute mobile code
  - engine written in IDLscript
  - [mobile] scripts written in IDLscript



Remote Scripting Engine: OMG IDL

```
interface Evaluation {
  typedef string Script; // A script is just a string!
   exception SyntaxError {};
   exception RuntimeError {string message;};
   exception IDLException {any exc;};

   Result evaluate(in Script a_script)
          raises(SyntaxError, RuntimeError, IDLException);
   . . .
};
```

- Other functionalities
  - session, remote variables, storage, ...

Remote Scripting Engine: Implementation

```
class EvaluationImpl
{ . . .
  proc evaluate(self, a_script)
  {
    try {
      return eval(script)
    } catch(CORBA.Exception e) {
      throw Evaluation.IDLException(e)
    } catch(SyntaxError e) {
      throw Evaluation.SyntaxError()
    } catch(e) {
      throw Evaluation.RuntimeError(e._toString())
    }
  }
}
```

Remote Scripting Engine: Example

```
unix> cssh client.cs file:/tmp/engine.ior
Remote Scripting Engine Shell 1.0
For instructions, type 'help'.
Engine> script
script> println("Hello World!")
script> return true
script>
Sends the script to the remote scripting::Engine.
output> Hello World!
The result is CORBA::Boolean(true)
script>
```

CorbaScript

Presentation of the first implementation of IDLscript specification

CorbaScript: the First Implementation of IDLscript

- LIFL provides an IDLscript implementation
  - based on CORBA 2.2
- Current supported ORB:
  - MICO, OAK, ORBacus, Visibroker 3.2 and 3.3 Orbix 2000, TAO, OmniORB 2
- Current supported OS:
  - AIX, HP-UX, Linux, SGI IRIX, Solaris, and Windows 95/98/NT
- Free and available with its full C++ source code
  - demos, naming and event services
- At: **http://corbaweb.lifl.fr/CorbaScript/**

Two Implementations

- IDLscript built with C++
  - Named CorbaScript
  - Access to C dynamic libraries
- IDLscript built with Java
  - Named JIDLscript
  - Access to all the Java classes (GUI, databases, ...)
  - A kind of *Java-Script* interpreter

Conclusion

Summary

- CORBA 3.0 incorporates a CORBA Scripting Language Specification
- IDLscript provides
  - a simple object-oriented scripting language
  - a CORBA dynamic binding
  - OMG IDL type system integration
  - implementable and already implemented
- Moreover IDLscript is compliant with current RFP submissions as long as they are described with OMG IDL

References

- OMG W3 site
  - http://www.omg.org
- IDLscript specification
  - ftp://ftp.omg.org/pub/docs/ptc/99-12-05
- Our W3 site
  - http://corbaweb.lifl.fr
  - articles and CorbaScript implementation