CORBA Scripting Language Specification

Version 1.0
June 2001

PATENT

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

NOTICE

The information contained in this document is subject to change without notice. The material in this document details an Object Management Group specification in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification in any company's products.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR PARTICULAR PURPOSE OR USE. In no event shall The Object Management Group or any of the companies listed above be liable for errors contained herein or for indirect, incidental, special, consequential, reliance or cover damages, including loss of profits, revenue, data or use, incurred by any user or any third party. The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the issue reporting form at *http://www.omg.org/library/issuerpt.htm*.

Contents

Contents

Contents

Contents

Preface

About the Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by several hundred members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

About CORBA Language Mapping Specifications

The CORBA Language Mapping specifications contain language mapping information for the following languages:

- Ada
- C
- C++
- COBOL
- IDL Script
- IDL to Java
- Java to IDL
- Python
- Smalltalk

Each language is described in a separate stand-alone volume.

Alignment with CORBA

The following table lists each language mapping and the version of CORBA that this language mapping is aligned with.

Language Mapping	**Aligned with CORBA version**
Ada	CORBA 2.0
C	CORBA 2.1
C++	CORBA 2.3
COBOL	CORBA 2.1
IDL to Java	CORBA 2.3
Java to IDL	CORBA 2.3
Lisp	CORBA 2.3
IDL Script	CORBA 2.3
Smalltalk	CORBA 2.0

Associated OMG Documents

The CORBA documentation is organized as follows:

- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.

- *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
- *CORBA Services: Common Object Services Specification* contains specifications for OMG's Object Services.
- *CORBA Common Facilities*: contains services that many applications may share, but which are not as fundamental as the Object Services.
- CORBA domain specifications are comprised of stand-alone documents for each specification; however, they are listed under the domain headings, such as Telecoms, Finance, Med, etc.

OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote.

To obtain OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue, Suite 201
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Definition of CORBA Compliance

The minimum required for a CORBA-compliant system is adherence to the specifications in CORBA Core and one mapping. Each additional language mapping is a separate, optional compliance point. Optional means users aren't required to implement these points if they are unnecessary at their site, but if implemented, they must adhere to the *CORBA* specifications to be called CORBA-compliant. For instance, if a vendor supports C++, their ORB must comply with the OMG IDL to C++ binding specified in this manual.

Interoperability and Interworking are separate compliance points. For detailed information about Interworking compliance, refer to the *Common Object Request Broker: Architecture and Specification*, *Interworking Architecture* chapter.

As described in the *OMA Guide*, the OMG's Core Object Model consists of a core and components. Likewise, the body of *CORBA* specifications is divided into core and component-like specifications. The structure of this manual reflects that division.

The *CORBA* specifications are divided into these volumes:

1. The *Common Object Request Broker: Architecture and Specification*, which includes the following chapters:
   - **CORBA Core**, as specified in Chapters 1-11
   - **CORBA Interoperability**, as specified in Chapters 12-16

- **CORBA Interworking**, as specified in Chapters 17-21

2. The Language Mapping Specifications, which are organized into the following stand-alone volumes:
   - **IDL Scripting Language**
   - **Mapping of OMG IDL to the Ada Programming Language**
   - **Mapping of OMG IDL to the C Programming Language**
   - **Mapping of OMG IDL to the C++ Programming Language**
   - **Mapping of OMG IDL to the COBOL Programming Language**
   - **Mapping of OMG IDL to the Java Programming Language**
   - **Mapping of Java Programming Language to OMG/IDL**
   - **Mapping of OMG IDL to the Smalltalk Programming Language**

Acknowledgments

The following companies submitted and/or supported parts of this specification:

- AIRSYS ATM
- Alcatel
- Commissariat à l'Energie Atomique
- INRIA
- Institut National des Télécommunications
- Laboratoire d'Informatique Fondamentale de Lille
- Object–Oriented Concepts, Inc.
- Silicomp Ingenierie
- Spacebel
- Université de Nantes - LRSG

IDLscript Overview 1

Note – The CORBA Scripting Language specification is aligned with CORBA version 2.3.

Contents

This chapter contains the following sections.

1.1 Scripting Languages

A scripting language simplifies the access and the use of computer system resources like files and processes in the context of an operating system shell, relational database query requests in the context of SQL, and graphic widgets in the context of Tcl/Tk. These resources are used without the need to write complex programs, hence the following benefits:

- **Simplicity of use**: A script is often easier to write and more concise (no variable declarations, dynamic typing, garbage collector) than its equivalent written in a standard programming language. The simplicity of scripting languages allows users, even novices, to develop small scripts that meet their needs.

- **Easy to learn**: The “teachability” of a scripting language is often more simple than a “traditional” language like C++. The training time is shorter for a scripting language.
- **Enhanced productivity**: This ease of use makes development easier and more flexible, as the user can prototype scripts in interactive mode, then use them in batch processing mode. This also encourages the exchange of scripts between users: they can adapt them to meet their individual needs.
- **Reduced cost**: Simplicity and productivity respectively mean reduced training costs for users and reduced operating costs in conventional computer environments.

However this previous list is not exhaustive and does not capture all scripting benefits.

1.2 CORBA and Scripting Languages

These benefits can be applied to a CORBA environment by providing a binding between scripting languages and OMG IDL. This considerably improves the ability to make use of CORBA distributed objects during all of the development, implementation, and execution steps:

- **Design and prototyping**: During the design step of a distributed CORBA application, two important problems may occur: the choice of OMG IDL interfaces and the choice of object distribution. Currently there is no miraculous solution to these two problems, only experience and know-how allow selecting the “right” choices. Under these conditions it is necessary to be able to prototype quickly in order to evaluate fundamental choices. But prototyping in a compiled language such as C++ implies a complex and costly development cycle, hence the advantage of using an interpreted language with a short development cycle in order to develop functional models.
- **Development and testing**: During the development of an object-oriented client/server application using CORBA, developers must write a number of pieces of programs in order to check the validity and the operation correctness of their CORBA objects. These test codes are hard to debug and write due to the complexity of mapping rules. In addition, they become useless when the components are correctly implemented. In this context, a command interpreter saves a lot of time and effort. It becomes possible to immediately and interactively test object implementations during development. In addition, object test codes can be generated automatically from the Interface Repository and data on interface semantics, resulting in automated testing.
- **Configuration and administration**: Most of the services and object frameworks require a number of client programs to configure, administrate, and connect the objects (such as the Naming Service). This large number of client programs often depends on the number of operations described in the objects’ OMG IDL interfaces. A dynamic scripting language then becomes an excellent alternative for supporting the multitude of programs as they can be written using a few instructions and evolve rapidly to meet the needs of service administrators.

- **Using components**: Experienced users can design scripts themselves to meet their own specific needs. In this way, using components available from the ORB, they can extract relevant data without the need to refer to ORB specialists.
- **Assembling software components**: Scripts can be used to assemble existing components in order to create new ones. The new components encapsulate all of the functions of connected components and provide new functions. Therefore we obtain a kind of “software glue” to build new objects by aggregating existing objects. In addition, these new components can be used from CORBA applications just like ordinary objects.
- **Evolution**: If the components evolve or if new ones appear, using scripts means that it is possible to discover them dynamically at execution time and therefore to use them as soon as they become available. Minor OMG IDL modifications do not necessarily require rewriting scripts.

Therefore a scripting language can offer a number of services during the life cycle of an object-oriented distributed service. The various uses imply that the scripting language provides the necessary mechanisms for discovering, invoking, and navigating among CORBA objects and for implementing objects using scripts. Navigating in and using large graphs of disparate objects imply dynamically acquiring the stubs of the types encountered as the scripting language cannot know ahead of time all of the OMG IDL types.

1.3 The IDLscript Language

IDLscript is a new general purpose object-oriented scripting language dedicated to CORBA that allows any user to develop their activities by simply and interactively accessing objects available on the ORB. Therefore the user is completely free to operate, administrate, configure, connect, create, and delete distributed objects on the ORB.

The binding between CORBA and IDLscript is achieved through the DII and the Interface Repository. The DII is used to construct requests at runtime and the IFR is used to check parameters types of requests (also at runtime). Moreover, using the DSI, IDLscript allows one to implement OMG IDL interfaces through scripted objects. Figure 1-1 on page 1-4 illustrates the IDLscript architecture.

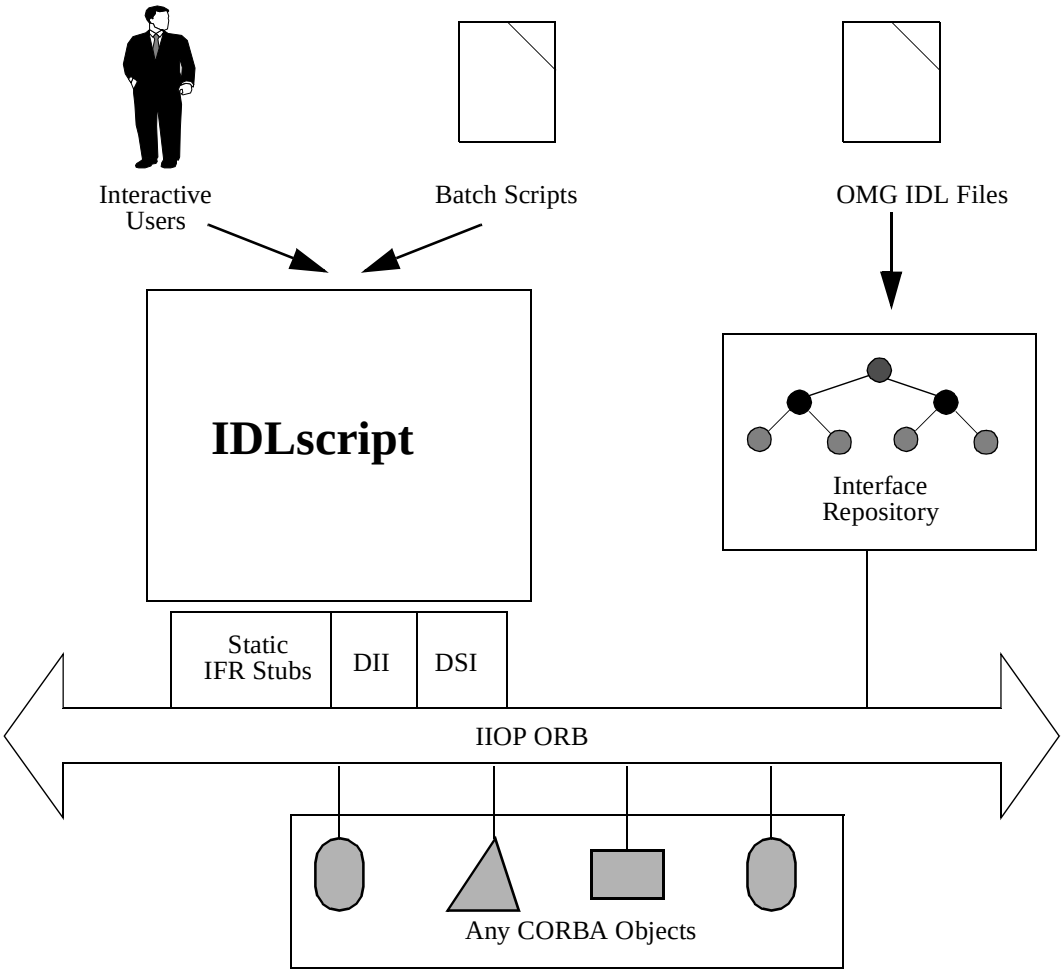

Figure 1-1 The IDLscript Architecture

The main features of IDLscript described in Chapters 2 and 3 include:

- **Interpretation:** The IDLscript engine is a scripting interpreter. It provides three execution modes: the interactive one, the batch one, and the embedded one. In the first mode, users provide their scripts interactively. In the second one, the interpreter loads and executes file scripts allowing batch processing or server implementations. In the last one, the interpreter can be embedded in another program and then interprets strings as scripts.
- **General purpose:** IDLscript is a true high level language comprising programming concepts such as structured procedures, modularity, and object-oriented programming (classes/instances, multiple inheritance, and polymorphism). The IDLscript language provides various syntactical constructions such as basic values and types (integer, double, boolean, character, string, array, and dictionary),

expressions (arithmetic, relational, and logical operators), assignments, control flow statements, procedures, classes, modern exception handling (throw/try/catch/finally) and modules (downloadable scripts).

- **Object-oriented:** All scripting values are encapsulated by internal engine objects. These objects provide some attributes and methods according to their type. The dotted notation is used to access/modify object attributes (i.e., *variable = object.attribute*, *object.attribute = value*) and invoke object methods (i.e., *object.method(parameters)*). IDLscript also allows the definition of scripting classes.
- **Dynamic typing:** A scripting value/object is the instance of one type. Types are also objects. A type can be a subtype of several other ones. Coercion rules are defined between types. This defines a type conformity tree used for runtime type checking. For example, method parameter type controls and automatic operand coercions (e.g., *10 + 3.14*). Moreover, scripts can dynamically access to the type conformity tree to check explicitly the type of an object (i.e., any object has a *_type* attribute and an *_is_a* method).
- **Reflexivity:** The IDLscript engine allows the introspection of any scripting object (values and types). The introspection encompasses object displaying and dynamic attribute, method and type discovering.
- **Adaptability:** IDLscript is a powerful scripting framework which can be adapted to meet users' needs. This framework can be extended by new internal classes which implement new object types. For example, an extension allows scripts to access to any Java class or object through the Java Virtual Machine.
- **Dynamic CORBA binding:** The integration between IDLscript and the ORB is fully dynamic: there is no stubs/skeletons generation. The IDLscript engine discovers OMG IDL specifications through the Interface Repository. When scripts invoke CORBA objects, the Dynamic Invocation Interface and the Dynamic Skeleton Interface are internally used to send and receive requests and the IFR is used to check parameter types at runtime. But users never use directly these CORBA dynamic mechanisms: they are totally hidden by the scripting engine.
- **Complete OMG IDL binding:** All OMG IDL concepts such as basic types, modules, constants, enumerations, structures, unions, typedefs, sequences, arrays, interfaces, exceptions, TypeCode, and Any types are directly and transparently available to scripts. The user must only give the IDL scoped name of accessed IDL specifications. These IDL concepts are reflected by scripting objects which are implemented by the scripting engine. Reflexivity is available on all these objects. Scripts can display any IDL values or definitions. Users can interactively discover the content of an IDL module or interface, the signature (parameters and exceptions) of an IDL operation, the mode and type of an IDL attribute, or the definition of a complex IDL type (enum, array, sequence, struct, union, and typedef).
- **Object binding:** To access and invoke CORBA objects, users must know their CORBA object references. IDLscript proposes several ways to obtain these references. Users can specify a known object network address described with the OMG's IOR format or with an ORB-specific URL format (i.e., IP host, IP port, and a local implementation object name). Moreover, standard CORBA Name and/or

Trader services can be dynamically used to obtain needed users' object references. To obtain these services, the standard ORB operations are available. Obtained object references are automatically narrowed to the most derived IDL interfaces.

- **Dynamic invocation:** IDLscript allows scripts to invoke IDL operations, access IDL attributes of remote CORBA objects/components. All type checks and coercions/conversions are automatically done by the interpreter. Parameter coercions are automatically done according to IDL signatures. IDLscript provides a simple Java-like exception mechanism that allows one to catch users' defined IDL exceptions and also standard CORBA system exceptions. CORBA requests are sent by the Dynamic Invocation Interface.
- **Dynamic implementation:** CORBA objects (and components and listeners) are implemented by scripting classes. Incoming requests are intercepted by the Dynamic Skeleton Interface and are forwarded to scripting objects. The scripting engine automatically converts incoming/outcoming IDL values to/from scripting objects respectively.

1.4 An IDLscript Example

This section presents a simple IDLscript example: a distributed grid application. This example aims at presenting the usefulness and simplicity of the new IDLscript language: access to any OMG IDL specifications, connection to any CORBA objects, access to OMG IDL attributes, invocation of OMG IDL operations, handling of OMG IDL exceptions, and finally implementation of CORBA objects and servers.

1.4.1 A Grid Distributed Application

As this example is an illustration of IDLscript, the object model of this application is deliberately simplified. This application is composed of a **Factory** OMG IDL interface that allows the creation of **Grid** objects:

```
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);
    };
};
```

A grid is a matrix of values (the Value type definition). The Coord structure defines matrix positions and dimensions. The InvalidCoord exception handles out of matrix bounds. The **Grid** interface provides the **dimension** attribute which returns the matrix dimension and operations to **get** and **set** values. The **destroy** operation allows clients to destroy a **Grid** object. The **Factory** interface provides the **create_grid** operation to create new grids. This operation creates a grid with the related dimension and initializes each item of the matrix. All OMG IDL type and interface definitions of this application are defined into the **GridService** OMG IDL module.

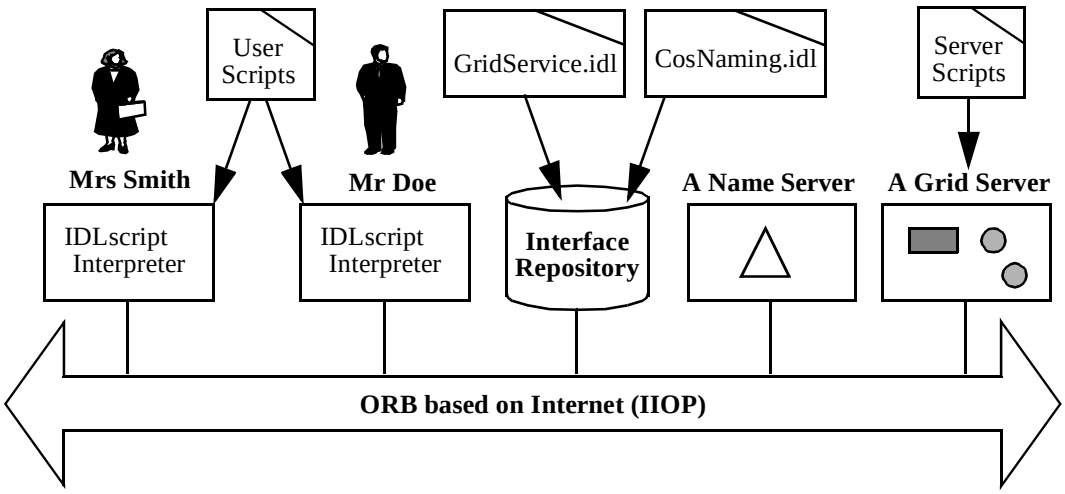

Figure 1-2 The Distributed Grid Application

Figure 1-2 shows the runtime distribution of this application. The **Grid** server contains a **GridService::Factory** CORBA object and the set of grid objects created by this factory. This server is composed of a set of IDLscript scripts that implement the OMG IDL interfaces of the **GridService** module and the server main function. The factory object reference is registered into the standard CORBA Name Service to allow client applications to retrieve it. In this example, the Interface Repository only contains the OMG IDL specifications of used CORBA objects, here the **GridService.idl** and **CosNaming.idl** OMG IDL files. Through this type information, an IDLscript interpreter can have access to all **CosNaming::NamingContext**, **GridService::Factory**, and **GridService::Grid** objects connected to the ORB. Finally, Mrs. Smith and Mr. Doe, end-users or CORBA specialists, can interactively act on the available CORBA objects thanks to the IDLscript interpreter. Moreover, they can share user scripts that provide advanced processes on CORBA objects.

1.4.2 Basic Functionalities

To perform the users' activities presented in Section 1.2, "CORBA and Scripting Languages," on page 1-2, IDLscript is a true high level language comprising programming concepts such as structured procedures, modularity, and object-orientation (classes/instances, multiple inheritance, and polymorphism). IDLscript is a script interpreter shell:

```
unix_promt> cssh
CorbaScript 1.3.1 for ORBacus 3.1.3 for C++
Copyright 1996-99 LIFL, France
>>>

unix_promt> cssh a_script_file.cs
```

IDLscript can be used from the command line (interactively) or in batch processing mode using script files. A script is a set of instructions such as; display a value, call-up an operation, assign a variable, control flows, and handle exceptions. This language supports a few basic data types wired into the interpreter: integers, strings, arrays, associate tables, basic OMG IDL, data types, etc. The conformity of expressions is checked dynamically at execution time using a dynamic typing mechanism. As IDLscript is object-oriented, all values are objects. The dotted notation is used to express operation call-up, attributes access, or modification. Moreover, IDLscript provides standard algorithm constructions (variables, tests, loops) used to express complex scripts.

1.4.3 Dynamic CORBA Connection

When a user invokes a CORBA object, the interpreter checks that the parameter types are conformed to the OMG IDL specifications contained in the Interface Repository. Invocations are performed via the Dynamic Invocation Interface. In addition, OMG IDL interfaces can be implemented using IDLscript classes. The IDLscript interpreter then uses the Dynamic Skeleton Interface to intercept and decode the requests sent to the objects implemented by scripts.

1.4.4 Direct Access to all OMG IDL Definitions

Through IDLscript, users can interactively and transparently access any OMG IDL specifications contained in the Interface Repository. This allows one to discover OMG IDL interfaces, operation parameters and exceptions, the fields in a structure, or the content of a module. The user must only give the scoped name of accessed OMG IDL specifications as presented here:

```
GridService.Grid
   OMG-IDL interface GridService::Grid {
      attribute readonly struct Coord dimension;
      void set (in struct Coord pos, in Value val)
         raises(GridService::InvalidCoord);
      Value get (in struct Coord pos)
         raises(GridService::InvalidCoord);
      void destroy ();
};

GridService.Coord
   OMG-IDL struct Coord {
```

```
        unsigned short x;
        unsigned short y;
};
```

IDLscript is transparently connected to the Interface Repository and accesses any OMG IDL definitions loaded into the Interface Repository as shown in Figure 1-2 on page 1-7.

1.4.5 Connection to any CORBA Object

To access and invoke CORBA objects, users must know their CORBA object references. IDLscript proposes several ways to obtain these references. Users can specify a known object network address described with the OMG's IOR format or with the Interoperable Name Service URL format (i.e., **`iioploc://`** and **`iiopname://`**). Moreover, standard CORBA Name and/or Trader services can be used to obtain users' needed object references. To obtain these services, the **`list_initial_services`** and **`resolve_initial_references`** operations from the **`CORBA::ORB`** interface are directly available. Consider the following examples:

```
factory = GridService.Factory("IOR:00000000000001c4...")
factory = GridService.Factory(
            "iioploc://an_IP_host_name:5000/factory")
CORBA.ORB.list_initial_services ()
["InterfaceRepository", "NameService", "TradingService",...]
NS = CORBA.ORB.resolve_initial_references("NameService")
      factory = NS.resolve ( [ ["aGridService", ""] ] )
```

In the last way, the user does not need to specify the type of the returned object. The IDLscript interpreter refers to the Interface Repository to determine the interface for the accessed objects and then checks the typing of invocations. When a CORBA request returns an object reference, IDLscript automatically creates an object reference for the dynamic type of the returned object. If the interpreter does not yet know the **`GridService.Factory`** type, it automatically loads its definition into its local Interface Repository cache. Therefore, users can navigate through the naming service graph and discover at execution time the type of visited objects.

1.4.6 OMG IDL Operations, Attributes, and Exceptions

As illustrated in the **resolve** operation invocation, the user does not have to specify the parameter types sent to the operations as IDLscript automatically performs the conversions. The **`[["aGridService", ""]]`** value is an array that contains an array with two items. This value is automatically converted into a **CosNaming::Name,** which is an OMG IDL sequence of **CosNaming::NameComponent** structures containing two OMG IDL string fields and then it is forwarded to the **resolve** operation.

```
grid = factory.create_grid ([20,5], 1)
# or more precisely, (GridService.Coord(20,5),
GridService.Value(1))
```

```
grid.dimension
GridService::Coord(20,5)
    try {
        grid.set([100,100],10)
    } catch (GridService::InvalidCoord e) {
    println ("GridService::InvalidCoord raises on ", e.pos)
    }
GridService::InvalidCoord raises on
GridSevice::Coord(100, 100)
```

The previous example illustrates the simplicity of IDLscript to invoke OMG IDL operations, access OMG IDL attributes of remote CORBA objects. All type checks and conversions are automatically done by the interpreter. Moreover, IDLscript provides a simple Java-like exception mechanism that allows scripts to catch user defined OMG IDL exceptions and also standard CORBA system exceptions.

1.4.7 Procedures and Modules

Naturally, these previous scripts are very rudimentary but IDLscript allows the storage of more ambitious scripts using procedures and modules. The procedures are used to capture users' reusable scripts. The returned result and procedure parameters are not typed. These procedures can be grouped in downloadable modules.

The following script fragment is part of the **`gridTools`** module. This module contains a procedure (**`DisplayGrid`**) that iterates on a grid to obtain matrix values by calling the **`get`** OMG IDL operation and display them. The user can therefore download the **`gridTools`** module to access this procedure and then execute it on the grid object previously obtained. Declarations contained in an IDLscript module are accessible with the dotted notation.

```
# File: gridTools.cs
proc DisplayGrid (grid)
{
   dim = grid.dimension
   h = dim.y
   w = dim.x
   println ("The dimensions of this grid are ", w, "*", h)
   # iterate to get each values of the grid
   for i in range (0, h-1) {
   for j in range (0, w-1) {
      print (' ', grid.get([i,j]))
   }
      println ()
   }
}

import gridTools
gridTools.DisplayGrid(grid)
The dimensions of this grid are 20*5
   1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

In this way a number of users' activities can be implemented without the need for the user to be a CORBA expert. It is still necessary to know the IDLscript language and the object OMG IDL interfaces to access them. But writing IDLscript scripts appears far easier than writing CORBA programs in a compiled language. Users can rapidly meet their specific needs and exchange scripts when their activities have points in common.

1.4.8 Implementation of OMG IDL Interfaces

A script handles local values and remote CORBA objects, acting just like a pure CORBA client program. Another IDLscript functionality supports the implementation of new object types (local or CORBA ones). It integrates object concepts such as classes, multiple inheritance, and polymorphism. Instance methods are grouped into classes and must take an explicit first parameter that refers to the receiver instance. However there is no enforced convention name for this parameter: users can choose any name like self, this, or anything else. Through this instance reference, the method codes can access instance attributes. Instance attributes are declared at their first assignment.

```
# File: grid_impl.cs
class GRID {
   # GRID instance initialization
   proc __GRID__ (self, dim, init_value)
   { # This GRID instance (self) is a
   GridService.Grid object
      CORBA.ORB.connect (self, GridService.Grid)
      # set the GRID instance attributes
      self.dim = dim
      self.values = create_matrix (dim, init_value)
   }

   # Creation of a matrix
   proc create_matrix (dim, init_value)
   {
      w=dim.x
      l=dim.y
      values = array.create(w)
      for i in range(0,w-1) {
      tmp = array.create(l)
      for j in range(0,l-1) { tmp[j] = init_value }
      values[i] = tmp
   }
   return values
   }
```

```
   # Implementation of the GridService::Grid interface
   # implements the readonly 'dimension' attribute
   proc _get_dimension (self)
   {
      return self.dim
   }
   # implements the 'set' operation
   proc set (self, pos, val)
   {
      try {
         self.values[pos.y][pos.x] = val
         } catch (BadIndex exc) {
         throw GridService.InvalidCoord(pos)
      }
   }

   # implements the 'get' operation
   proc get (self, pos)
   {
      try {
         return self.values[pos.y][pos.x]
         } catch (BadIndex exc) {
         throw GridService.InvalidCoord(pos)
      }
   }
   # implements the 'destroy' operation
   proc destroy (self)
   {
      CORBA.ORB.disconnect (self)
   }
}

class FACTORY
{
   proc __FACTORY__ (self)
   {
      CORBA.ORB.connect (self, GridService.Factory)
   }

   # the 'create_grid' operation
   proc create_grid (self, dim, init_value)
   {
      grid = GRID(dim, init_value)
      return grid._this
   }
}
```

The previous code presents an implementation of the Grid service. The **GRID** and **FACTORY** classes implement respectively the **GridService::Grid** and **GridService::Factory** interfaces. IDLscript enforces a convention name for the

instance initialization method (**__GRID__** and **__FACTORY__**). The OMG IDL operations are implemented by instance methods with the same name. The OMG IDL attributes are also implemented by instance methods called by the attribute name prefixed by **_get_** for the attribute getting and by the **_set_** prefix for the attribute setting.

The **CORBA.ORB** symbol refers to the IDLscript reflection of the ORB object. This object provides operations to connect/disconnect class instances to/from a CORBA object reference. The **connect** operation allows one to associate an IDLscript instance to a new CORBA object: the first parameter refers to the instance and the second one refers to the OMG IDL interface that the instance implements. A third optional parameter allows user to explicitly set the key. The **disconnect** operation cuts this association, then all its CORBA object references become invalid.

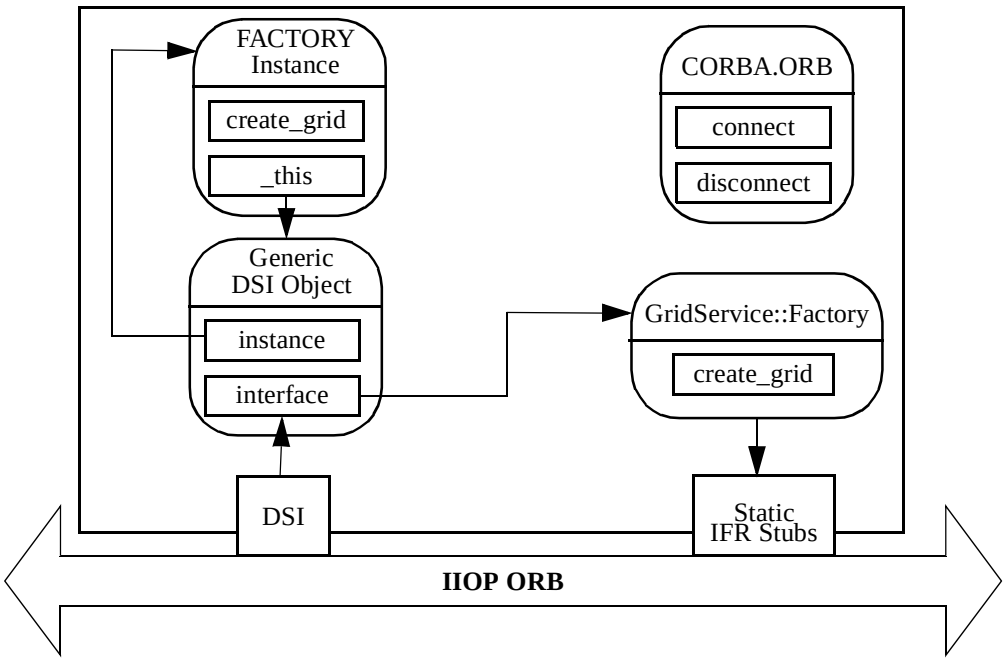

Figure 1-3 The Grid Server Objects Architecture

Figure 1-3 presents the IDLscript objects architecture after the creation of the **FACTORY** instance. The **GridService::Factory** object is in the local cache of the OMG IDL interface. This cache communicates with the Interface Repository to obtain OMG IDL type information. The generic DSI object is connected to the ORB to receive requests for the **FACTORY** instance. Received requests are checked thanks to the local cache (**interface**) and if they are correct, then they are forwarded to the **FACTORY** instance (**instance**). This instance implements the OMG IDL operations and attributes of the **GRIDService::Factory** interface. The **_this** instance attribute refers to the generic DSI object. It is used when the object must give its CORBA object reference.

This approach is similar to the TIE approach used in C++ and Java mappings. This mechanism of request delegation allows one to provide several DSI object references for the same IDLscript instance: several OMG IDL interfaces could be implemented by a single IDLscript instance.

1.4.9 Creation of Stand-alone CORBA Servers

In this way a script can become a CORBA object server accessible to all CORBA programs and therefore to other scripts. The following code shows the Grid server implementation:

```
# Load the GridService implementation `grid_impl.cs' file
import grid_impl

# Create a FACTORY instance
factory = grid_impl.FACTORY()

# Obtain the Name Service reference
NS = CORBA.ORB.resolve_initial_references("NameService")

# Register the server object into the Name Service
NS.bind ( [["aGridService", ""]], factory._this)

# Start the main loop to wait for ORB requests
CORBA.ORB.run ()

# Unregister the server object from the Name Service
NS.unbind ( [["aGridService", ""]])
```

This server script imports the previous Grid implementation module containing the **`GRID`** and **`FACTORY`** classes. It creates then a **`FACTORY`** instance and registers it into the standard CORBA Name service with the **`bind`** operation. Then this script starts a main loop to wait for ORB requests (**`CORBA.ORB.run`**). Finally, it unregisters the factory object from the Name Service (**`unbind`** operation) when the server is stopped or shutdowns.

1.4.10 Conclusion

This chapter has presented a quick tour of the IDLscript functionalities. IDLscript simultaneously offers enough syntax constructions and semantic entities such as expressions, numerous types of basic data, all of the types expressed in OMG IDL, the modules, the procedures, the classes and the instances in order to quickly develop client programs and CORBA object servers. In addition, the dynamic loading of modules is used to structure scripts into easily reusable entities. These entities are used to quickly write sets of procedures to use an application and reuse them to build a number of client applications, meeting the specific needs of each developer and each user in a CORBA environment.

The IDLscript Language Core 2

This chapter describes the IDLscript core language including lexical conventions, syntactical and semantic constructs.

Contents

This chapter contains the following sections.

2.1 Overview

IDLscript is a simple and powerful general purpose object-oriented scripting language. All the IDLscript entities are objects with attributes and methods. Moreover, IDLscript is dedicated to CORBA environments allowing users to write scripts to easily access to CORBA objects. Scripts can also implement CORBA objects (e.g., callback objects) via classes. However the information presented herein is fully CORBA and OMG IDL independent. The binding between IDLscript and CORBA is presented in the next chapter.

The IDLscript lexical rules are very similar to OMG IDL rules, although keywords and punctuation characters are different to support programming concepts. The description of IDLscript's lexical conventions is presented in Section 2.2, "Lexical Conventions.

The IDLscript grammar provides a small and "easy-to-learn" set of constructs to define scripts, expressions, variables, control flow statements, procedures, classes, exceptions, and modules. The grammar is described below in Section 2.3, "IDLscript Grammar," on page 2-6.

The IDLscript core concepts are respectively presented in Section 2.4, "Scripts," on page 2-9, Section 2.5, "Expressions," on page 2-9, Section 2.7, "Objects and Types," on page 2-15, Section 2.8, "Control Flow Statements," on page 2-23, Section 2.9, "Procedures," on page 2-26, Section 2.10, "Classes," on page 2-28, Section 2.11, "Exceptions," on page 2-32, and Section 2.12, "Modules," on page 2-36.

Scripts can be interactively provided by users or stored into source files with the ".is" extension.

The description of IDLscript grammar uses a syntax notation that is similar to Extended Backus-Naur Format (EBNF). Table 2-1 lists the symbols used in this format and their meaning.

Table 2-1 IDLscript Symbols and Meanings

Symbol	Meaning
::=	Is defined to be
|	Alternatively
<text>	Nonterminal
"text"	Literal
*	The preceding syntactic unit can be repeated zero or more times
+	The preceding syntactic unit can be repeated one or more times
{}	The enclosed syntactic units are grouped as a single syntactic unit
[]	The enclosed unit is optional -- may occur zero or one time

2.2 Lexical Conventions

This section[1] presents the lexical conventions of IDLscript. It defines tokens in an IDLscript script and describes comments, identifiers, keywords, and literals such as integer, floating point, and character constants and string literals.

As OMG IDL, IDLscript uses the ISO Latin-1 (8859.1) character set. This character set is divided into alphabetic characters (letters), digits, graphic characters, the space (blank) character and formatting characters (for more information, see Table 3-2, Table 3-3, Table 3-4, and Table 3-5 in the CORBA 2.3 specification).

2.2.1 Tokens

There are five kinds of tokens: identifiers, keywords, literals, operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments, as described below, are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to be the longest string of characters that could possibly constitute a token.

2.2.2 Comments

The sharp character (#) starts a comment, which terminates at the end of the line on which it occurs. Comments may contain alphabetic, digit, graphic, space, horizontal tab, vertical tab, and form feed characters. The following example illustrates comments:

```
>>> # This is a comment
```

2.2.3 Identifiers

Identifiers refer to names of variables, types, procedures, classes, and modules. An identifier is an arbitrarily long sequence of alphabetic, digit, and underscore ("_") characters. The first character must be an alphabetic or underscore character. All characters are significant. The following examples are valid identifiers:

```
identifier  identifier123  an_identifier  An_Identifier
```

Note that IDLscript is a case sensitive language: **`an_identifier`** and **`An_Identifier`** are two different identifiers.

---

1. This section is an adaptation of *The CORBA 2.3 Specification*, Chapter 3, already an adaptation of Ellis, Margaret A. and Bjarne Stroustrup, *The Annoted C++ Reference Manual*, Addison-Wesley Publishing Company, Reading, Massachussets, 1990, ISBN 0-201-51459-1, Chapter 2. It differs in the list of legal keywords and punctuation.

2.2.4 Keywords

The identifiers listed in Table 2-2 are reserved for use as keywords and may not be used otherwise.

Table 2-2 Keywords

`catch`	`class`	`del`	`do`	`else`
`finally`	`for`	`if`	`import`	`in`
`proc`	`return`	`throw`	`try`	`while`

Keywords obey the rules for identifiers (see Section 2.5.3, “Identifiers,” on page 2-11) and must be written exactly as shown in the above list. For example, “ class” is correct; “Class” refers to an identifier and can produce an interpretation error.

IDLscript scripts use the characters shown in Table 2-3 as punctuation.

Table 2-3 Punctuation Characters

`(`	`)`	`[`	`]`	`{`	`}`	`,`	`;`	`.`	`::`	`:`
`+`	`-`	`*`	`/`	`%`	`\`	`!`	`&&`	`||`		
`=`	`==`	`!=`	`<`	`<=`	`>`	`>=`				

2.2.5 Literals

This section describes the following literals:

- Integer
- Floating-point
- Character
- String

2.2.5.1 Integer Literals

An integer literal consisting of a sequence of digits is taken to be decimal (base ten) unless it begins with 0 (digit zero). A sequence of digits starting with 0 is taken to be an octal integer (base eight). The digits 8 and 9 are not octal digits. A sequence of digits preceded by 0x or 0X is taken to be a hexadecimal integer (base sixteen). The hexadecimal digits include a or A through f or F with decimal values ten through fifteen, respectively. For example, the number twelve can be written 12, 014, or 0xC.

2.2.5.2 Floating-point Literals

A floating-point literal consists of an integer part, a decimal point, a fraction part, an e or E, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of decimal (base ten) digits. Either the integer part or the fraction part (but not both) may be missing; either the decimal point or the letter e (or E) and the exponent (but not both) may be missing. Consider the following examples:

```
3.  3.2  .2  3.2e-4  .2e15  10e10
```

2.2.5.3 *Character Literals*

A character literal is one or more characters enclosed in single quotes, as in the following examples:

```
'a' '\t' '\045' '\x4f'
```

A character is an 8-bit quantity with a numerical value between 0 and 255 (decimal). The value of a space, alphabetic, digit, or graphical character literal is the numerical value of the character as defined in the ISO Latin-1 (8859.1) character set standard (See Table 3-2 on page 3-4, Table 3-3 on page 3-4, and Table 3-4 on page 3-5 in the CORBA 2.3 specification). The value of null is 0. The value of a formatting character literal is the numerical value of the character as defined in the ISO 646 standard (see Table 3-5 on page 3-6 in the CORBA 2.3 specification). The meaning of all other characters is implementation-dependent.

Nongraphic characters must be represented using escape sequences as defined below in Table 2-4. Note that escape sequences must be used to represent single quote and backslash characters in character literals.

Table 2-4 Escape Sequences

Description	Escape Sequence
newline	\n
horizontal tab	\t
vertical tab	\v
backspace	\b
carriage return	\r
form feed	\f
alert	\a
backslash	\\
question mark	\?
single quote	\'
double quote	\"
octal number	\ooo
hexadecimal number	\xhh

If the character following a backslash is not of those specified, the behavior is undefined. An escape sequence specifies a single character.

The escape \ooo consists of the backslash followed by one, two, or three octal digits that are taken to specify the value of the desired character. The escape \xhh consists of the backslash followed by x followed by one or two hexadecimal digits that are taken to specify the value of the desired character. A sequence of octal or hexadecimal digits

is terminated by the first character that is not an octal digit or a hexadecimal digit, respectively. The value of a character constant is implementation dependent if it exceeds that of the largest char.

Wide character and wide string literals are specified exactly like character and string literals. All character and string literals, both wide and non-wide, may only be specified (portably) using the characters found in the ISO 8859-1 character set, that is identifiers will continue to be limited to the ISO 8859-1 character set.

2.2.5.4 String Literals

A string literal is a sequence of characters (as defined in Section 2.2.5.3, “Character Literals,” on page 2-5) surrounded by double quotes, as in the following examples:

```
"Hello world!\n" "An \"embedded\" string"
```

Adjacent string literals are concatenated. Characters in concatenated strings are kept distinct. For example,

```
"\xA" "B"
```

contains the two characters ’\xA’ and ’B’ after concatenation (and not the single hexadecimal character ’\xAB’).

The size of a string literal is the number of character literals enclosed by the quotes, after concatenation. The size of the literal is associated with the literal. Within a string, the double quote character must be preceded by a \.

A string literal may not contain the character ’\0’.

2.3 IDLscript Grammar

```
(1)          <script>  ::= <statements>
(2)      <statements>  ::= <statement>*
(3)       <statement>  ::= ";"
                        |   "{" <statements> "}"
                        |   <expression>
                        |   <variable_management>
                        |   <control_flow_statements>
                        |   <procedure_declaration>
                        |   <class_declaration>
                        |   <exception_management>
                        |   <module_management>
(4)      <expression>  ::= <literal>
                        |   <identifier>
                        |   "(" <expression> ")"
                        |   <arithmetic_expression>
                        |   <relational_expression>
                        |   <logical_expression>
                        |   <procedural_call>
```

```
                        |   <attribute_get>
                        |   <method_call>
                        |   <array_creation>
                        |   <dictionary_creation>
                        |   <indexed_get>
(5)        <literal>  ::= <long_literal>
                        |   <double_literal>
                        |   <character_literal>
                        |   <string_literal>
(6) <arithmetic_expression>
                        ::= "+" <expression>
                        |   "-" <expression>
                        |   <expression> "+" <expression>
                        |   <expression> "-" <expression>
                        |   <expression> "*" <expression>
                        |   <expression> "/" <expression>
                        |   <expression> "%" <expression>
                        |   <expression> "\" <expression>
(7) <relational_expression>
                        ::= <expression> "==" <expression>
                        |   <expression> "!=" <expression>
                        |   <expression> "<"  <expression>
                        |   <expression> "<=" <expression>
                        |   <expression> ">"  <expression>
                        |   <expression> ">=" <expression>
(8) <logical_expression>
                        ::= "!" <expression>
                        |   <expression> "&&" <expression>
                        |   <expression> "||" <expression>
(9) <procedural_call> ::= <identifier> "(" <arguments> ")"
(10)     <arguments>   ::= [ <expression_list> ]
(11) <expression_list> ::= <expression> { "," <expression>}*
(12)  <attribute_get>  ::= <expression> "." <identifier>
                       |   <expression> "!" <identifier>
(13)    <method_call>  ::= <expression> "." <identifier>
                           "(" <arguments> ")"
                       |   <expression> "!" <identifier>
                           "(" <arguments> ")"
(14) <array_creation>  ::= "[" <arguments> "]"
(15) <dictionary_creation>
                        ::= "{" <dictionary_expression_list>
                            "}"
(16) <dictionary_expression_list>
                        ::= [ <dictionary_expression> { ","
                            <dictionary_expression> }* ]
(17) <dictionary_expression>
                        ::= <expression> ':' <expression>
(18) <indexed_get>  ::= <expression> "[" <expression> "]"
(19) <variable_management>
                        ::= <assignment_statement>
                        |   <del_statement>
```

```
(20) <assignment_statement>
                        ::= <identifier> "=" <expression>
                        |   <expression> "." <identifier>
                            "=" <expression>
                        |   <expression> "!" <identifier>
                            "=" <expression>
                        |   <expression> "[" <expression> "]"
                            "=" <expression>
(21)  <del_statement>  ::= "del" <identifier>
                        |   "del" <expression> "."
                            <identifier>
(22) <control_flow_statements>
                        ::= <if_statement>
                        |   <while_statement>
                        |   <do_statement>
                        |   <for_statement>
                        |   <return_statement>
(23)   <if_statement>  ::= "if" "(" <expression> ")"
                            <statement>
                            [ "else" <statement> ]
(24) <while_statement> ::= "while" "(" <expression> ")"
                            <statement>
(25)   <do_statement>  ::= "do" <statement>
                            "while" "(" <expression> ")"
(26)  <for_statement>  ::= "for" <identifier> "in"
                            <expression> <statement>
(27) <exception_management>
                        ::= <throw_statement>
                        |   <try_catch_finally_statement>
(28) <throw_statement> ::= "throw" <expression>
(29) <try_catch_finally_statement>
                        ::= "try" "{" <statements> "}"
                            { "catch" "(" <exception_type>
                            <identifier> ")"
                            "{" <statements> "}" }*
                            [ "catch" "(" <identifier> ")"
                            "{" <statements> "}" ]
                           [ "finally" "{" <statements> "}" ]
(30) <exception_type>  ::= <identifier> { "." <identifier> }*
(31) <return_statement>
                        ::= "return" [ <expression> ]
(32) <procedure_declaration>
                        ::= "proc" <identifier> "("
                            [ <formal_parameter_list> ] ")"
                            "{" <statements> "}"
(33) <formal_parameter_list>
                        ::= <identifier_list> { ","
                            <identifier> "=" <expression> }*
(34) <identifier_list> ::= <identifier> { ‘,’ <identifier>
                            }*
(35) <class_declaration>
```

```
                              ::= "class" <identifier> [ "("
                                  <inherited_class_list> ")" ]
                                  "{" <statements> "}"
(36) <inherited_class_list>
                              ::= <expression_list>
(37) <module_management>
                              ::= <import_module>
(38)  <import_module>  ::= "import" <identifier_list>
```

2.4 Scripts

An IDLscript script consists of zero or more statements. A statement can be a null statement (';'), a statement block surrounded by bracket characters ('{' and '}'), an expression, a variable management statement, a control flow statement, a procedure declaration, a class declaration, an exception management statement, or a module management statement. The syntax is:

```
<script> ::= <statements>
<statements> ::= <statement>*
<statement> ::= ";"
            |   "{" <statements> "}"
            |   <expression>
            |   <variable_management>
            |   <control_flow_statements>
            |   <procedure_declaration>
            |   <class_declaration>
            |   <exception_management>
            |   <module_management>
```

See Section 2.5, "Expressions," on page 2-9, Section 2.6, "Variable and Attribute Management," on page 2-14, Section 2.8, "Control Flow Statements," on page 2-23, Section 2.9, "Procedures," on page 2-26, Section 2.10, "Classes," on page 2-28, Section 2.11, "Exceptions," on page 2-32, and Section 2.12, "Modules," on page 2-36, respectively, for specifications of **<expression>**, **<variable_management>**, **<control_ flow_statements>**, **<procedure_declaration>**, **<class_declaration>**, **<exception_management>**, and **<module_management>**.

2.5 Expressions

This section describes the syntax for IDLscript expressions. These syntactical constructs are general and can be applied on any IDLscript objects. Their semantic meaning depends on the object's type as described in Sect i on2.7, "Objects and Types," on page 2-15.

2.5.1 Syntax

An IDLscript expression can be a literal, an identifier, a parenthetical expression, an arithmetic expression, a relational expression, a logical expression, a procedural call, an attribute getting, a method call, an array creation, a dictionary creation, and an indexed getting. The syntax is:

```
<expression> ::= <literal>
             |   <identifier>
             |   "(" <expression> ")"
             |   <arithmetic_expression>
             |   <relational_expression>
             |   <logical_expression>
             |   <procedural_call>
             |   <attribute_get>
             |   <method_call>
             |   <array_creation>
             |   <dictionary_creation>
             |   <indexed_get>
```

See Section 2.5.2, “Literal Values,” on page 2-10, Section 2.2.3, “Identifiers,” on page 2-3, Section 2.5.4, “Arithmetic Operators,” on page 2-11, Section 2.5.5, “Relational Operators,” on page 2-12, Section 2.5.6, “Logical Operators,” on page 2-12, Section 2.5.7, “Procedural Call,” on page 2-13, Section 2.5.8, “Attribute Getting,” on page 2-13, Section 2.5.9, “Method Call,” on page 2-13, Section 2.5.10, “Array Creation,” on page 2-14, Section 2.5.11, “Dictionary Creation,” on page 2-14, and Section 2.5.12, “Indexed Getting,” on page 2-14, respectively, for specifications of <literal>, <identifier>, <arithmetical_expression>, <relational_expression>, <logical_expression>, <procedural_call>, <attribute_get>, <method_call>, <array_creation>, <dictionary_creation>, and <indexed_get>.

2.5.2 Literal Values

The syntax for expression literals is:

```
<literal> ::= <long_literal>
          |   <double_literal>
          |   <character_literal>
          |   <string_literal>
```

Here, **<long_literal>**, **<double_literal>**, **<character_literal>**, and **<string_literal>** refers respectively to integer, float-point, character, and string lexical literals defined in Sectio n2.2.5, “Literals,” on page 2-4. Consider the following examples:

```
>>> 10                  # a long value
10
>>> 3.1415              # a double value
3.1415
>>> 'c'                 # a character value
```

```
'c'
>>> "Hello World!"      # a string value
"Hello World!"
```

Note that when IDLscript is interactively used it displays the result of the last expression evaluation.

2.5.3 Identifiers

Expression identifiers are defined as lexical identifiers described in Section 2.5.3, “Identifiers,” on page 2-11. These identifiers refer to named IDLscript objects like constants, variables, types, procedures, classes, modules, etc. The two predefined identifiers **`true`** and **`false`** respectively refer to IDLscript constant objects that represent the two boolean values. The **`Void`** identifier refers to the unique void object value. Consider the following examples:

```
>>> true                # the boolean true value
true
>>> false               # the boolean false value
false
>>> Void
>>>
```

Note that if an expression evaluation returns the **`Void`** value, then this value is not displayed.

2.5.4 Arithmetic Operators

The syntax for arithmetic expressions is:

```
<arithmetic_expression> ::= "+" <expression>
                          |   "-" <expression>
                          |   <expression> "+" <expression>
                          |   <expression> "-" <expression>
                          |   <expression> "*" <expression>
                          |   <expression> "/" <expression>
                          |   <expression> "%" <expression>
                          |   <expression> "\" <expression>
```

IDLscript supports the usual arithmetic operators: the "`+`" and "`-`" unary ones, and the "`+`", "`-`", "`*`", "`/`", and "`%`" binary ones. The "`\`" binary operator represents the integer division. Automatic needed value coercions are done for binary operators. These operators have the usual meaning. Consider the following examples:

```
>>> 10 + 3
13
>>> 10 - 3.3
6.7
>>> 10 / 3
3.33333
```

```
>>> 10 % 3    # only for long integers
1
>>> 10 \ 3    # only for long integers
3
```

2.5.5 Relational Operators

The syntax for relational expressions is:

```
<relational_expression> ::= <expression> "==" <expression>
                        |   <expression> "!=" <expression>
                        |   <expression> "<"  <expression>
                        |   <expression> "<=" <expression>
                        |   <expression> ">"  <expression>
                        |   <expression> ">=" <expression>
```

Relational operators are the classical binary ones: "==", "!=", "<", "<=", ">", and ">=". They return boolean values and operand type coercions are done automatically if needed. This also implies dynamic value type checking at execution time. These operators have the usual meaning. Consider the following examples:

```
>>> 10 == 3
false
>>> 3.1415 > 3
true
```

2.5.6 Logical Operators

The syntax for logical expressions is:

```
<logical_expression> ::= "!" <expression>
                     |   <expression> "&&" <expression>
                     |   <expression> "||" <expression>
```

Logical operators are the classical unary and binary ones. The unary *not* is represented by "!". The binary *and* is represented by "&&". The binary *or* is represented by "||". They take two boolean operands. These operators return a boolean value. Dynamic operand type checking is done at execution time. These operators have the usual meaning. Consider the following examples:

```
>>> ( 10 != 3.3 ) && true
true
>>> ( 10 < 3 ) || false
false
>>> true && false
false
>>> false || ( 10 > 3 )
true
>>> ! ( 10 == 3 )
true
```

2.5.7 Procedural Call

The syntax for procedural calls is:

```
<procedural_call> ::= <identifier> "(" <arguments> ")"
<arguments> ::= [ <expression_list> ]
<expression_list> ::= <expression> { "," <expression> }*
```

A procedural call can be applied to any IDLscript object named by an identifier. Arguments, surrounded by brackets, are composed of zero or more expressions separated by comma characters. The number of arguments and the meaning of a procedural call depend on the IDLscript object designed by the identifier. For example, if the object is a procedure (see Section 2.9, “Procedures,” on page 2-26), then the meaning is to execute this object procedure; whereas if the object is a class (see Section 2.10, “Classes,” on page 2-28), then the meaning is the instantiation of this class.

2.5.8 Attribute Getting

The syntax for attribute getting is:

```
<attribute_get> ::= <expression> "." <identifier>
                |   <expression> "!" <identifier>
```

An attribute getting can be applied to any expression object. The identifier names the accessed attribute. Two point notations are provided: ’.’ and ’!’. The meaning of an attribute getting depends on the target object and the used point notation. For most objects, these two notations are equivalent and the meaning is the access to an existing attribute of the target object. However applied to a CORBA object reference (see Section 3.12.3, “Object References,” on page 3-22), the meaning is a synchronous or a deferred attribute getting.

2.5.9 Method Call

The syntax for method calls is:

```
<method_call> ::= <expression> "." <identifier>
                     "(" <arguments> ")"
                |   <expression> "!" <identifier>
                     "(" <arguments> ")"
```

A method call can be applied to any expression object. The identifier names the invoked method. Method arguments, surrounded by brackets, are composed of zero or more expressions separated by comma characters. Two point notations are also provided: ’.’ and ’!’. The meaning of a method call depends on the target object and the used point notation. For most objects, these two notations are equivalent and the meaning is the invocation of an existing method of the target object. However applied to a CORBA object reference (see Section 3.12.3, “Object References,” on page 3-22), the meaning is a synchronous or a deferred method call.

2.5.10 Array Creation

The syntax for array creations is:

```
<array_creation> ::= "[" <arguments> "]"
```

At creation time, an array object (see Section 2.7.4, “Array Objects,” on page2-19) can be initialized with zero or more expression objects. Consider the following examples:

```
>>> [ ]                              # an empty array
[]
>>> [ 1, 2.3, ’c’, "hello", true ] # an heterogeneous array
[ 1, 2.3, ’c’, "hello", true]
```

2.5.11 Dictionary Creation

The syntax for dictionary creations is:

```
<dictionary_creation> ::= "{" <dictionary_expression_list>
                          "}"
<dictionary_expression_list> ::= [ <dictionary_expression>
                           { "," <dictionary_expression> }* ]
<dictionary_expression> ::= <expression> ’:’ <expression>
```

At creation time, a dictionary object (see Section 2.7.5, “Dictionary Objects,” on page 2-21) can be initialized with zero or more key/value expression pairs separated by commas. The key and the value of a pair is separated by ’:’. Consider the following example:

```
>>> { 1: "one", 2: "two", 3: "three" }
{ 1: "one", 2: "two", 3: "three"}
```

2.5.12 Indexed Getting

The syntax for indexed getting is:

```
<indexed_get> ::= <expression> "[" <expression> "]"
```

An indexed getting can be applied to any expression object. The accessed index is also an expression object. The meaning depends on the target object.

2.6 Variable and Attribute Management

This section describes the syntax for variable and attribute management, that is assignment and deletion constructs. The syntax is:

```
<variable_management> ::= <assignment_statement>
                        |   <del_statement>
```

2.6.1 Assignments

The syntax for assignments is:

```
<assignment_statement> ::= <identifier> "=" <expression>
    |     <expression> "." <identifier> "=" <expression>
    |     <expression> "!" <identifier> "=" <expression>
    | <expression> "[" <expression> "]" "=" <expression>
```

The first construct is dedicated to variable assignments. Variables can refer to any expression object. They are defined at their first assignment. During execution time, a variable can take different kinds of values. Consider the following examples:

```
>>> v = 10
>>> v
10
>>> v = "Hello"
"Hello"
```

Other constructs are for attribute and indexed assignments. Their meaning depends on the target object.

2.6.2 The Del Statement

The syntax for deletions is:

```
<del_statement> ::= "del" <identifier>
                  |   "del" <expression> "." <identifier>
```

The **`del`** statement construct allows scripts to forget a previous defined variable. The variable is designed by the **`identifier`**. Note that this identifier can be preceded by an **`expression`** that defines the scope of the variable such as a module, a class, or an instance.

2.7 Objects and Types

This section describes the main IDLscript object types and their functionalities.

2.7.1 Everything is Typed Object

As IDLscript is an object-oriented scripting language, all scripted entities such as literals, arrays, dictionaries, procedures, classes, instances, exceptions, and modules are represented by objects. Each object provides a set of functionalities: operators, attributes, and methods. These functionalities are used through the syntactical constructs presented in Section 2.5, “Expressions,” on page 2-9.

The set of functionalities of an object is defined by its type. Through this type, the interpreter checks the validity of every operator, attribute, and method call. When a typing error occurs, the interpreter throws an internal exception (see Section 2.11.1, “Internal Exceptions,” on page 2-32). Moreover, types are also IDLscript objects. The

_type attribute allows scripts to access the IDLscript type of any object. It allows programmers to check typing information (for example to check argument types of a procedure). Table 2-5 enumerates the set of functionalities that are supported by all IDLscript objects and types.

Table 2-5 The Object and Type Functionalities

Functionality	Explanation
object._type	Returns the type object of any *object*.
object._is_a(type)	Returns *true* if the *object* is of a certain *type* or of a type which is a subtype of this *type*.
object._toString()	Returns a string that is the textual representation of an *object*.
type._type	Returns the meta type of any *type* object.
type1._is_a(type2)	Returns *true* if the *type1* is equal or is a subtype of *type2*.
type._toString()	Returns a string that is the textual representation of the *type* object.

2.7.2 Basic Value Types

The basic object types are accessible through **boolean**, **long**, **double**, and **char** identifiers. Consider the following examples:

```
>>> b = true
>>> b._type
< type boolean ... >
>>> b._is_a(boolean)
true
>>> b._is_a(long)
false
>>> b._toString()
"true"
>>> l = 10
>>> l._type
< type long ... >
>>> l._is_a(long)
true
>>> l._is_a(double)
false
>>> l._toString()
"10"
>>> d = 3.1415
>>> d._type
< type double ... >
>>> d._is_a(double)
true
```

```
>>> d._is_a(char)
false
>>> d._toString()
"3.1415"
>>> c = 'c'
>>> c._type
< type char ... >
>>> c._is_a(char)
true
>>> c._is_a(boolean)
false
>>> c._toString()
"c"
```

These types provide the classical semantic for operators (see Section 2.5.4, “Arithmetic Operators,” on page 2-11, Section 2.5.5, “Relational Operators,” on page 2-12, and Section 2.5.6, “Logical Operators,” on page 2-12) and automatic coercions.

2.7.3 String Objects

The **`string`** identifier refers to the string type. Strings support a set of attributes, methods, and operators. All these functionalities are enumerated in Table 2-6 and they never modify the target string. When indexes are out of the string bounds, an exception BadIndex is raised (see Section 2.11.1, “Internal Exceptions,” on page 2-32).

Table 2-6 The String Type Functionalities

Functionality	Explanation
`s.length`	Returns the length of the *s* string.
`s[i]`	Returns the character at the *i* position. The index ranges from 0 to s.length - 1.
`c + s`	Returns the concatenation of the *c* character and the *s* string.
`s + c`	Returns the concatenation of the *s* string and the *c* character.
`s1 + s2`	Returns the concatenation of the *s1* and *s2* strings.
`s1 == s2`	Returns *true* if *s1* contains the same sequence of characters as *s2*.
`s1 != s2`	Returns *true* if *s1* contains a different sequence of characters as *s2*.
`s1 < s2`	Returns *true* if *s1* is lexicographically lower than *s2*.
`s1 <= s2`	Returns *true* if *s1* is lexicographically lower or equal to *s2*.
`s1 > s2`	Returns *true* if *s1* is lexicographically greater than *s2*.
`s1 >= s2`	Returns *true* if *s1* is lexicographically greater or equal to *s2*.
`s.index(c)`	Returns the position of the first occurrence of the *c* character or -1 if *c* does not occur.
`s.index(c,pos)`	Returns the position of the first occurrence of the *c* character starting the search at the *pos* index or -1 if *c* does not occur.

Table 2-6 The String Type Functionalities

Functionality	Explanation
`s1.index(s2)`	Returns the position of the first occurrence of the *s2* string or -1 if *s2* does not occur.
`s1.index(s2,pos)`	Returns the position of the first occurrence of the *s2* string starting the search at the *pos* index or -1 if *s2* does not occur.
`s.rindex(c)`	Returns the position of the last occurrence of the *c* character or -1 if *c* does not occur.
`s.rindex(c,pos)`	Returns the position of the last occurrence of the *c* character starting the backward search at the *pos* index or -1 if *c* does not occur.
`s1.rindex(s2)`	Returns the position of the last occurrence of the *s2* string or -1 if *s2* does not occur.
`s1.rindex(s2,pos)`	Returns the position of the last occurrence of the *s2* string starting the backward search at the *pos* index or -1 if *s2* does not occur.
`s.substring(bi)`	Returns a new string that is a substring of *s* beginning at the *bi* index.
`s.substring(bi,ei)`	Returns a new string that is a substring of *s* between the *bi* and *ei* indexes.
`s.toLowerCase()`	Returns a new string that is a lower case copy of the *s* string.
`s.toUpperCase()`	Returns a new string that is a upper case copy of the *s* string.

Consider the following examples:

```
>>> s = "Hello World!"
>>> s._type
< type string ... >
>>> s._is_a(string)
true
>>> s._is_a(boolean)
false
>>> s._toString()
"Hello World!"
>>> s.length
12
>>> s[1]
'e'
>>> s + '!'
"Hello World!!"
>>> "Hello " + "World!"
"Hello World!"
>>> s == "Hello World!"
true
>>> s.index('o')
4
>>> s.index('o',6)
```

```
7
>>> s.index("l")
2
>>> s.index("l",5)
9
>>> s.substring(3,7)
"lo Wo"
>>> s.toLowerCase()
"hello world!"
>>> s.toUpperCase()
"HELLO WORLD!"
```

2.7.4 Array Objects

The **`array`** identifier refers to the array type. Arrays are dynamically extensible containers of any IDLscript objects. Arrays are built using '`[`' and '`]`' delimiters and values are separated by commas ('`,`'). Array elements can have different types. Arrays can be embedded in other arrays. Moreover, array objects provide a set of operators, attributes, and methods. All these functionalities are enumerated in Table 2-7. When indexes are out of the array bounds, an IDLscript internal exception BadIndex is raised.

Table 2-7 The Array Type Functionalities

Functionality	Explanation
`a.length`	Returns to the length of the *a* array.
`a[i]`	Returns the value at the *i* position. The index ranges from 0 to *a.length* - 1.
`a[i] = v`	Updates the component value at the *i* position. The index ranges from 0 to *a.length* - 1.
`a1 + a2`	Returns a new array which is the concatenation of the *a1* and *a2* arrays.
`a.append(v)`	Appends the *v* object at the end of the *a* array.
`a.insert(v,i)`	Inserts the *v* object at the *i* position. The index ranges from 0 to *a.length*.
`a.delete(i)`	Deletes the component value at the *i* position. The index ranges from 0 to *a.length* - 1.
`a.remove(v)`	Removes the first occurrence of the *v* object. Returns *true* if *v* occurs.
`a.contains(v)`	Returns *true* if the *v* value is contained in the array.
`a.index(v)`	Returns the position of the first occurrence of the *v* object or -1 if *v* does not occur.
`a.index(v,pos)`	Returns the position of the first occurrence of the *v* object starting the search at the *pos* index or -1 if *v* does not occur.

Table 2-7 The Array Type Functionalities

Functionality	Explanation
`a.rindex(v)`	Returns the position of the last occurrence of the *v* object or -1 if *v* does not occur.
`a.rindex(v,pos)`	Returns the position of the last occurrence of the *v* object starting the backward search at the *pos* index or -1 if *v* does not occur.
`array.create(n)`	Creates an array initialized with *n Void* objects.

Consider the following examples:

```
>>> # heterogeneous array
>>> a = [ true, [1, 3.1415], 'c', "Hello World!"]
>>> a._type
< type array ... >
>>> a._type == array
true
>>> a._is_a(boolean)
false
>>> a._toString()
"[ true, [1, 3.1415], 'c', "Hello World!"]"
>>> a.length
4
>>> a[1]
[1, 3.1415]
>>> a[1] = 10
>>> a
[ true, 10, 'c', "Hello World!"]
>>> a + [1,2]
[ true, 10, 'c', "Hello World!", 1, 2]
>>> a.append (false)
>>> a
[ true, 10, 'c', "Hello World!", false]
>>> a.insert("a value", 1)
>>> a
[ true, "a value", 10, 'c', "Hello World!", false]
>>> a.delete(2)
>>> a
[ true, "a value", 'c', "Hello World!", false]
>>> a.remove("a value")
true
>>> a
[ true, 'c', "Hello World!", false]
>>> a.contains(10)
false
>>> a.index(false)
3
>>> a.index(true, 1)
-1
```

```
>>> a = [ true, 'c', 10, 'c', false]
>>> a.rindex('c')
3
>>> a.rindex('c',2)
1
>>> a = array.create(5)
>>> a
[ Void, Void, Void, Void, Void]
```

2.7.5 Dictionary Objects

The **dictionary** identifier refers to the dictionary type. A dictionary object is a powerful container to store any key - value associations such as indexed tables, structured records, etc. Keys and values are of any IDLscript object types. Dictionaries are built using '{' and '}' delimiters, associations are separated by commas (','), and key and value by the ':' character. Dictionary objects provide a set of operators, attributes and methods. All these functionalities are enumerated in Table 2-8. Searching a key that is not contained by a dictionary raises an IDLscript internal NotFound exception.

Table 2-8 The Dictionary Type Functionalities

Functionality	Explanation
dict.size	Returns the number of associations in the *dict* dictionary.
dict.keys	Returns an array of the key objects in the *dict* dictionary.
dict.values	Returns an array of the value objects in the *dict* dictionary.
dict[key]	Returns the value associated to the *key* in the *dict* dictionary.
dict[key] = value	Updates the *value* associated to a *key* or adds this *key* - *value* association in the *dict* dictionary.
dict.contains(value)	Returns *true* if the *value* is associated to a key in the *dict* dictionary.
dict.containsKey(key)	Returns *true* if the *key* is present in the *dict* dictionary.
dict.remove(key)	Removes the *key* and its corresponding value from the *dict* dictionary.

Consider the following examples:

```
>>> d = { 1: "one", 2: "two", 3: "three"}
>>> d._type
< type dictionary ... >
>>> d._type == dictionary
true
>>> d._is_a(boolean)
false
>>> d._toString()
"{ 1: "one", 2: "two", 3: "three"}"
```

```
>>> d.size
3
>>> d.keys
[1, 2, 3]
>>> d.values
["one", "two", three"]
>>> d[1]
"one"
>>> d[4] = "four"
>>> d
{ 1: "one", 2: "two", 3: "three", 4: "four"}
>>> d.contains("two")
true
>>> d.containsKey(4)
true
>>> d.remove(2)
>>> d
{ 1: "one", 3: "three", 4: "four"}
```

2.7.6 Predefined Internal Procedures

IDLscript provides some predefined internal procedures respectively named by the following identifiers: **`eval`**, **`exec`**, **`getline`**, **`print`**, and **`println`**.

Table 2-9 The Predefined Internal Procedures

Internal Procedures	Explanation
`eval(string)`	Evaluates a *string* containing an IDLscript script.
`exec(string)`	Executes the file named by *string*.
`getline()`	Reads a text line from the standard input stream.
`print(arg1, ..., argn)`	Prints zero or more object arguments.
`println(arg1, ... argn)`	Prints zero or more object arguments and a new line.

The **`eval`** function provides the classical powerful evaluation function. It takes a stringified script, executes it, and returns the result of this evaluation. This allows programmers to construct interpretable IDLscript code at execution time.

The **`exec`** function executes a script file. Variables, procedures, and classes defined into the file are always available after the file execution.

The **`getline`** function allows scripts to read a text line from the standard input stream and returns a string containing this text line.

The interpreter automatically displays the last evaluated expression. But it can be necessary into complex scripts to display a value or a set of values at any time, for example, during a loop. The **print** procedure allows scripts to display a set of object expressions. The **println** procedure displays a new line after printing all the expressions.

These internal procedures are executed using the procedural calling notation. Consider the following examples:

```
>>> s = "1 + 1"
>>> eval (s)
2
>>> exec("a_script.cs")
. . .
>>> s = getline()
Hello World!
>>> s
"Hello World!"
>>> print (100, '\n', "string1 string2\n")
100
string1 string2
>>> println (1, ' ', 'c', ' ', true, ' ', "string")
1 c true string
```

2.8 Control Flow Statements

This section describes the syntax for IDLscript control flow statements.

2.8.1 Syntax

An IDLscript control flow statement can be an **if**, a **while**, a **do**, a **for**, and a **return** statement. The syntax is:

```
<control_flow_statements> ::= <if_statement>
                            |   <while_statement>
                            |   <do_statement>
                            |   <for_statement>
                            |   <return_statement>
```

See Section 2.8.2, “The `If` Statement,” on page 2-23, Section 2.8.3, “The `While` Statement,” on page 2-24, Section 2.8.4, “The `Do` Statement,” on page 2-24, Section 2.8.5, “The `For` Statement,” on page 2-25, and Section 2.8.6, “The `Return` Statement,” on page 2-25, respectively, for specifications of **<if_statement>**, **<while_statement>**, **<do_statement>**, **<for_statement>**, and **<return_statement>**.

2.8.2 The If Statement

The syntax for the **if** statement is:

```
<if_statement> ::= "if" "(" <expression> ")" <statement>
                   [ "else" <statement> ]
```

The **if** statement construct allows scripts to test a condition expression. If it is true, the following statement is executed, else the statement after **else** is executed. Of course, the **else** clause is optional.

The condition must be a boolean expression: a variable containing a boolean object, a relational operator (e.g., ==, !=, <, <=, > or >=) or a composition of boolean expressions (e.g., &&, || or !). The dynamic type of the expression is checked at runtime. Consider the following examples:

```
>>> i = 1
>>> if ( i == 1) println("i == 1");
i == 1
>>> i = 2
>>> if ( i == 1) { println("i == 1") }
    else { println("i != 1") }
i != 1
```

2.8.3 The While Statement

The syntax for the **while** statement is:

```
<while_statement> ::= "while" "(" <expression> ")"
                      <statement>
```

The **while** statement construct allows scripts to iterate a set of statements while a condition expression is true. The condition must be a boolean expression object and is checked at runtime. If the condition is false at the first time, the statements are never executed. Consider the following example:

```
>>> i = 0
>>> while ( i < 10 ) {
      print (i, ' ')
      i = i + 1
     }
0 1 2 3 4 5 6 7 8 9
```

2.8.4 The Do Statement

The syntax for the **do** statement is:

```
<do_statement> ::= "do" <statement>
                   "while" "(" <expression> ")"
```

The **do** statement construct allows scripts to iterate a set of statements while a condition expression is true. The condition must be a boolean expression object and is checked at runtime. Consider the following example:

```
>>> i = 0
>>> do {
      print (i, ' ')
      i = i + 1
    } while ( i < 10 )
0 1 2 3 4 5 6 7 8 9
```

2.8.5 The For Statement

The syntax for the **for** statement is:

```
<for_statement> ::= "for" <identifier> "in" <expression>
                            <statement>
```

The **for** statement construct allows scripts to iterate on an **expression** enumeration of objects. During each **statement** execution loop, the **identifier** variable contains the next object of the **expression**. The **expression** must be an enumerated object such as a string or an array. This property is checked at runtime. Consider the following examples:

```
>>> a = ["Monday", "Tuesday", "Wednesday", "Thursday", "Fri-
day", "Saturday", "Sunday"]
>>> for i in a print (i, ' ');
Monday Tuesday Wednesday Thursday Friday Saturday Sunday
>>> for i in "hello world!" print (i, ' ');
h e l l o   w o r l d !
>>> for i in range(0,9) print (i, ' ');
0 1 2 3 4 5 6 7 8 9
>>> r = range(9,0,-1)
>>> for i in r print (i, ' ');
9 8 7 6 5 4 3 2 1 0
```

The **range** expression allows scripts to perform a loop on an integer interval. The two first arguments define respectively the first and last integer values of the interval, the third optional argument sets the interval increment. By default, this increment is equal to **1**. Moreover, a range expression is also an IDLscript object, it can be stored into a variable.

2.8.6 The Return Statement

The syntax for the **return** statement is:

```
<return_statement> ::= "return" [ <expression> ]
```

The **return** statement construct allows a script to interrupt its execution before the end of the script code. It is mainly used in procedures or instance methods to return a result to the caller.

The returned value is optional. In this way, the **return** statement returns automatically the **Void** object. This construct can be used when procedures want to prematurely stop their execution without returning a value.

2.9 Procedures

This section describes the syntax for IDLscript procedures.

2.9.1 Declaration

The syntax for procedure declarations is:

```
<procedure_declaration> ::= "proc" <identifier> "("
                            [ <formal_parameter_list> ] ")"
                            "{" <statements> "}"
<formal_parameter_list> ::= <identifier_list> { ","
                            <identifier> "=" <expression> }*
<identifier_list> ::= <identifier> { ‘,’ <identifier> }*
```

The **proc** declaration construct allows scripts to create a procedure. A procedure is specified by an **identifier** name and a list of formal parameters (**formal_parameter_list**) defined between brackets (’(’ and ’)’). A procedure body is composed of a set of **statements** between brackets (’{’ and ’}’). Consider the following example which declares a **sample** procedure with two formal parameters (**p1** and **p2**):

```
>>> proc sample (p1, p2)
    {
      println ("The 'sample' procedure is called with p1=",
               p1, " and p2=", p2)
    }
>>> sample (true,"hello")
The 'sample' procedure is called with p1=true and p2=hello
```

Procedures can be redefined at any time. The new procedure must only use the same name. The previous procedure version becomes unavailable.

2.9.2 Formal Parameters and Default Values

Formal parameters are not typed and there is no limit about their number. A default value can be assigned to the last formal parameters. These values are evaluated at the procedure creation time. The procedure statements can access directly to formal parameters as local variables. Consider the following example:

```
>>> proc display (p1, p2="World")
    {
      println (p1, ' ', p2, ’!’)
    }
>>> display ("Hello")
```

```
Hello World!
>>> display ("Hello", "You")
Hello You!
```

Formal parameters can be used in read and write mode inside the procedure; it does not affect the real parameter since procedures do not call update methods on the formal parameters.

2.9.3 The Returned Object

Procedures can return an object computed inside them using the **`return`** statement, and this stops the procedure execution. Consider the following example that presents a recursive implementation of a factorial function:

```
>>> proc fac (i)
    {
      if ( i == 1 ) return 1
      return i * fac (i - 1)
    }
>>> fac (5)
120
```

2.9.4 Local and Global Variables

Local variables can be defined anywhere inside a procedure. They are defined at their first assignment. If a local variable has the same name as a global variable, then this global variable is hidden in the procedure. Unhidden global variables can be accessed by procedures only in read mode. However global variables can be accessed and updated by prefixing them with the *global* scope name. Consider the following example:

```
>>> x = 5
>>> proc sample ()
    {
      # access to the global 'x' variable.
      println ("x=", x)
      x = 3                    # create a local 'x' variable.
      # access to the local 'x' variable.
      println ("x=", x)
      # access and update the global 'x' variable.
      global.x = global.x * 2
    }
>>> sample ()
x=5
x=3
>>> x
10
```

2.9.5 Procedure Aliasing

Procedures are objects, they can be assigned to a variable and be called using the new name. Consider the following example:

```
>>> alias = fac
>>> alias (5)
120
```

As procedures are objects, they can be transmitted to another procedure as a parameter. The following example illustrates passing procedures as parameters: the **sort_criteria** parameter of the **sort** procedure:

```
>>> proc down (a, b) { return a < b }
>>> proc up (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
          }
    }
>>> t = [ 60 , 6543 , 4 , 1124 , 1 ]
>>> sort (t)
>>> t
[1 , 4 , 60 , 124 , 6543]
>>> sort (t, down)
>>> t
[6543 , 124 , 60 , 4 , 1 ]
```

This **sort** procedure works with all basic IDLscript value types. By default, it uses the **up** function as sort criteria, but it is possible to pass another procedure like **down**. Note that the modification of an array item stays after the execution of a procedure, because an array is an object passed by reference.

2.10 Classes

The IDLscript language allows one to design script classes. IDLscript uses the classical functionalities of object-oriented programming. A class can define instance attributes, instance methods, class attributes, and class methods. Polymorphism, overriding, and multiple inheritance are available, but as scripts are not syntactical typed, overloading is not provided.

2.10.1 Declaration

The syntax for the **class** declaration is:

```
<class_declaration> ::= "class" <identifier>
                        [ "(" <inherited_class_list> ")" ]
                        "{" <statements> "}"
<inherited_class_list> ::= <expression_list>
```

The **class** declaration construct allows scripts to declare a class named by **identifier**. A class can inherit a set of parent classes (**inherited_class_list**). Finally, the class body is composed of a set of statements.

The class body statements define instance methods, class methods, and class attributes. Instance attributes are declared at their first assignment. The IDLscript class construct is very simple because we think that creating scripted objects must be as simple as possible.

2.10.2 A Simple Class Example

The following example shows a simple class that implements two dimensional points. This class illustrates the definition of instance attributes, instance methods, class attributes, and class methods.

```
>>> 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
    }
```

2.10.2.1 Instance Methods

In IDLscript, each instance method must have an explicit first argument (**self** for instance) that refers to the instance receiving the method call. However this first argument can have any name. Next arguments receive parameters of the method call.

It is possible to define an instance initialization method which is called at the class instantiation time (__**Point2D**__). This method must have the same name as the class and must be surrounded by two underscores (_).

2.10.2.2 Instance Attributes

Instance attributes are dynamically declared at their first assignment like in **self.x = x** and **self.y = y** statements of the initialization __**Point2D**__ method.

Instance methods can access directly to instance attributes just by prefixing them with the instance reference like in the **show** and **move** instance methods.

2.10.2.3 Class Methods

Any procedure declared in the scope of a class is considered as a class method like **how_many**.

2.10.2.4 Class Attributes

Class attributes are just variables assigned in the scope of a class like **nb_created_points**. Accessing to class attributes requires that they should be prefixed by their class name.

2.10.2.5 Class Instantiations

The procedural calling notation is used to instantiate a class. In IDLscript, there is no **new** keyword because everything is an object dynamically created. Consider the following example:

```
>>> p = Point2D(1,1)
>>> p
< Point2D instance
     x = 1
     y = 1
>
```

The first statement creates a **Point2D** instance. The second statement illustrates how IDLscript simply evaluates an instance: it shows the type of the instance and all instance attributes.

2.10.2.6 Instance Method Invocations

Consider the following example that illustrates method invocations on a **Point2D** instance:

```
>>> 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;
} >
```

The classical dotted notation is used to invoke instance methods. As other IDLscript objects, instances support the **_type** attribute which returns its instantiation class. The evaluation of a class shows the signatures of all instance methods, class methods, and class attributes.

2.10.3 A Single Class Inheritance Example

IDLscript provides a simple class inheritance mechanism. This allows a class to inherit other classes like in the following example where the class **Point3D** inherits the class **Point2D**. Overriding is available as shown by the **show** and **move** instance methods. Note that the polymorphism will not work if the procedure signature is changed by adding new parameters, IDLscript does not provide overloading. Moreover as procedures are IDLscript values, it is possible to define alias to access to inherited methods as shown by the **move2D** alias.

```
>>> class Point3D (Point2D) {
      proc __Point3D__ (self,x,y,z) {
        self.__Point2D__(x,y)
        self.z = z
      }
      proc show (self) { ... }
      move2D = Point2D.move
      proc move (self, p) {
        self.move2D (p.x, p.y)
        self.z = self.z + p.z
      }
    }
>>> p = Point3D(1,1,1)
```

2.10.4 A Multiple Class Inheritance Example

Multiple inheritance is available in IDLscript as shown by the following example where the class **ColoredPoint3D** inherits the **Point3D** and **ColoredPoint2D** classes.

```
>>> class ColoredPoint2D (Point2D) {
      proc __ColoredPoint2D__ (self,x,y,c) { ... }
      proc show (self) {...}
    }

>>> class ColoredPoint3D (Point3D, ColoredPoint2D) {
```

```
      proc __ColoredPoint3D__ (self,x,y,z,c) { ... }
      proc show (self) {...}
    }

>>> p = ColoredPoint3D(10,10,10,"green")
>>> p   < ColoredPoint3D instance
       x = 10
       y = 10
       z = 10
       c = "green"
```

The method lookup is based on the deep-first algorithm. So if a method has the same name in two inherited classes, it will be the version in the first class which will be chosen. Method aliasing allows one to simply change this standard method lookup.

2.10.5 Class and Instance Types

As classes and instances are IDLscript objects, they provide the standard attributes and methods to manipulate types (see Section 2.7.1, “Everything is Typed Object,” on page 2-15). Then type comparisons and dynamic type checking are simply available on classes and instances. Consider the following examples:

```
>>> ColoredPoint3D
< class ColoredPoint3D (Point3D,ColoredPoint2D) {
       proc __ColoredPoint3D__ (self, x, y, z, c);
       proc show (self);
   } >

>>> p._type == ColoredPoint3D
true
>>> p._type == Point2D
false
>>> p._is_a(Point2D)
true
>>> ColoredPoint3D._is_a(Point2D)
true
```

2.11 Exceptions

This section describes the IDLscript exception mechanism. There are two kinds of exceptions: internal interpreter exceptions and users’ script exceptions.

2.11.1 Internal Exceptions

The internal exceptions are used by the interpreter to signal syntax errors, bad type checkings, and invalid operations, or any other internal problems during the execution of a users’ script. Internal exceptions are listed below.

- BadArgumentNumber: This exception is thrown when a script calls a procedure or a method without passing enough parameters.
- BadIndex: This exception is thrown when the index to access a string (or an array) is out of the string (or array) bounds. If an index is less than zero or greater than the length of a string (or an array), then the interpreter throws this exception.
- BadTypeCoerce: This exception is thrown when a script tries to apply operations between incompatible types. For instance, adding a boolean with a string is impossible because the boolean and the string object cannot be coerced to two compatible objects, then the interpreter throws a bad type coerce exception. Moreover, this exception is thrown when parameters passed to an internal procedure are not compatible with formal parameter expected types.
- ExecutionStopped: This exception is thrown when the interpreter is stopped by an external reason like a **<CTRL-C>** signal.
- FileNotFound: This exception is thrown when a script tries to load another script of which the file name is unknown (or not understandable) by the underlying file system.
- NotFound: This exception is thrown when an undefined variable, an undefined attribute, or an undefined method is accessed by a script.
- NotImplemented: This exception is thrown when an internal IDLscript feature is not currently implemented.
- NotSupported: This exception is thrown when an operator or a syntactic construct is applied on an IDLscript object which does not support it.
- Overflow: This exception is thrown when the interpreter detects an arithmetic overflow.
- ReadOnlyAttribute: This exception is thrown when scripts try to affect a read only attribute.
- SyntaxError: This exception is thrown when a lexical or syntactic error appears in an interactive script, a downloaded script contained into a file, or a script evaluated by the **eval** function.

Consider the following examples:

```
>>> s = "Hello world!"

>>> s.toLowerCase(10)  # toLowerCase takes no parameter.
Exception: < BadArgumentNumber: < InternalMethod
string.toLowerCase() > needed = 0 given = 1 >
  File "stdin", line 1 in ?

>>> s[100]  # 100 is out of the string bounds.
Exception: < BadIndex: 100 must be between (0,11) on "Hello
world!" >
  File "stdin", line 1 in ?
```

```
>>> s < 10  # No type coercion between a string and a long
value.
Exception: < BadTypeCoerce: "Hello world!" < 10 >
  File "stdin", line 1 in ?

>>> while (true);  # an infinite loop.
Exception: < ExecutionStopped: by CTRL-C >
  File "stdin", line 1 in ?

>>> exec("a_script.cs")  # execute a script file not avail-
able.
Exception: < FileNotFound: 'a_script.cs' by exec() >
  File "stdin", line 1 in ?

>>> s1 # This is an undefined variable name.
Exception: < NotFound: variable 's1' >
  File "stdin", line 2 in ?

>>> s.an_attribute  # a string value does not provide this
attribute.
Exception: < NotFound: attribute 'an_attribute' in "Hello
world!" >
  File "stdin", line 1 in ?

>>> s(10)  # the procedural call construct is not available
on string values.
Exception: < NotSupported: call on "Hello world!" >
     File "stdin", line 1 in ?

>>> 10 \ 0  # division by zero.
Exception: < Overflow: divide by zero >
  File "stdin", line 1 in ?

>>> s.length = 10
Exception: < ReadOnlyAttribute: < InternalSlot readonly stri
ng.length > >
  File "stdin", line 1 in ?

>>> s.10  # this construction is not syntactically correct.
Exception: < SyntaxError before or on '10' >
     File "stdin", line 1 in ?
```

2.11.2 User Exceptions

Users can define their own exceptions. The exceptions are launched with the **`throw`** statement followed by an expression.

```
<throw_statement> ::= "throw" <expression>
```

Any IDLscript object can be used to throw a user exception. A script can throw a basic value such as a boolean, a long integer, a string, or a complex value like an array or a class instance.

```
>>> throw 10
Exception: < throw 10 >
     File "stdin", line 1 in ?

>>> throw "Hello"
Exception: < throw "Hello" >
     File "stdin", line 1 in ?

>>> throw [1,2]
Exception: < throw [1 , 2] >
     File "stdin", line 1 in ?

>>> class A_CLASS { proc __A_CLASS__(self,v) { self.v = v } }
>>> throw A_CLASS(1)
Exception: < throw < A_CLASS instance > >
     File "stdin", line 1 in ?
```

2.11.3 Exception Handling

Internal and user exceptions can be caught by scripts. The syntax for exception handling is:

```
<try_catch_finally_statement> ::= "try" "{" <statements> "}"
   { "catch" "(" <exception_type> <identifier> ")"
     "{" <statements> "}" }*
   [ "catch" "(" <identifier> ")" "{" <statements> "}" ]
   [ "finally" "{" <statements> "}" ]
<exception_type> ::= <identifier> { "." <identifier> }*
```

The **`try`** statement block surrounds a set of statements throwing exceptions. This block is followed by a set of **`catch`** statement blocks. Each **`catch`** block intercepts a type of exception values (**`exception_type`**). If the exception type is compatible with the type caught by a block, then the exception is stored into a variable (**`identifier`**) and the statements of this block are executed. The last and optional **`catch`** block (with no exception type) allows scripts to catch any exception. However, if the type of the current raised exception is not intercepted by a **`catch`** block, then this exception is thrown to the next encapsulating **`try`** block. Moreover, the optional **`finally`** block is executed in any case, this allows scripts to execute some statements if there are exceptions or not.

```
>>> proc exception_handling (v) {
      try {
        throw v
      } catch (boolean e) {
        println ("The exception is a boolean = ", e)
      } catch (long e) {
```

```
        println ("The exception is a long integer = ", e)
      } catch (string e) {
        println ("The exception is a string = ", e)
      } finally {
        println ("The finally block is executed.")
      }
    }

>>> exception_handling(true)
The exception is a boolean = true
The finally block is executed.

>>> exception_handling(1)
The exception is a long integer = 1
The finally block is executed.

>>> exception_handling("EXCEPTION")
The exception is a string = EXCEPTION
The finally block is executed.


>>> exception_handling([1, 2, 3])
The finally block is executed.
Exception: < throw [1 , 2 , 3] >
     File "stdin", line 3 in exception_handling
     File "stdin", line 1 in ?

>>> try {
      exception_handling(A_CLASS(1))
    } catch (e) {
      println ("The exception ", e, " is thrown by the proce-
dure.")
    }

The finally block is executed.
The exception < A_CLASS instance > is thrown by the proce-
dure.
```

2.12 Modules

Modules allow users to store reusable scripts into text files. This means that any text file containing IDLscript statements is a module. A module looks like an interactive script: it can declare variables, procedures, classes, and can execute any statements.

2.12.1 Importation

The syntax for module importations is:

```
<import_statement> ::= "import" <identifier_list>
<identifier_list> ::= <identifier> { ‘,’ <identifier> }*
```

To load modules in the interpreter, users must invoke the **import** statement with a list of one or more module names.

The file storing a module has the same name as the module postfixed by the **.cs** extension. The interpreter has to look for module files using an environment variable named **CSPATH**. This variable lists the directories containing module files. Directories are separated by ':' or ';' depending on operating systems.

2.12.2 Initialization

When a module is loaded for the first time, the interpreter executes all statements contained into the module file. Then, the module can declare any procedure or class, and execute any statements to initialize global module variables. Next importations do not reexecute the statements.

2.12.3 Access to the Content

The dotted notation is used to access variables, procedures, and classes of a module:

```
module_name.name_of_a_variable
module_name.name_of_a_procedure (parameters)
module_name.name_of_a_class
```

2.12.4 Module Aliasing

As all IDLscript entities, a module is an object that can be assigned to a variable and passed as a parameter to a procedure.

```
>>> import module1
>>> module2 = module1
>>> a_procedure(module2)
```

2.12.5 Module Management

The list of all the loaded modules is contained into the **sys.modules** scope. Consider the following example:

```
>>> sys.modules
< scope sys.modules {
    module module1;
  } >
>>> del sys.modules.module1
>>> sys.modules
< scope sys.modules {
  } >
```

The **del** statement can be applied to the **sys.modules** scope to explicitly delete a loaded module. The next importation of this deleted module reloads the module file and executes it.

The OMG IDL Binding 3

Contents

This chapter contains the following sections.

This chapter presents the binding between IDLscript and OMG IDL. It shows how all OMG IDL constructions such as basic types, modules, constants, enumerations, structures, unions, typedefs, sequences, arrays, interfaces, attributes, operations, exceptions, values, TypeCodes, and Anys are represented and can be manipulated from the IDLscript language.

3.1 Overview

IDLscript provides a dynamic IDL binding that allows users to access directly and naturally to any IDL specifications loaded into the Interface Repository. This approach does not need to generate stubs and skeletons; therefore, users can invoke, navigate, and discover any CORBA objects at runtime. IDLscript totally hides the complexity of the DII, DSI, and Interface Repository APIs, and it internally uses them to construct and receive requests in a safe way.

The IDLscript type system integrates seamlessly the OMG IDL type system. For each IDL construction, this chapter presents how to access the IDL definition, how it is represented with IDLscript, how to create such values, and how to manipulate them using the IDLscript language.

From Section 3.2, “Binding for Basic OMG IDL Types to Section 3.11, “Binding for OMG IDL Exception,” on page 3-16, this chapter presents the binding for basic elements of OMG IDL. Section 3.12, “Binding for OMG IDL Interface,” on page 3-21 presents the binding for OMG IDL interfaces and how to implement these interfaces using IDLscript (Section 3.13, “Implementing OMG IDL Interfaces,” on page 3-25). Section 3.14, “Binding for OMG IDL Value,” on page 3-29 presents the binding for OMG IDL values and how to implement them with IDLscript classes. TypeCode and Any are respectively presented in Section 3.16, “Binding for OMG IDL TypeCode,” on page 3-35 and Section 3.17, “Binding for OMG IDL Any,” on page 3-38. Finally the access to the heart of CORBA is presented in Section 3.18, “The Global CORBA Object,” on page 3-40.

3.2 Binding for Basic OMG IDL Types

In IDLscript, any item is accessible by an identifier; therefore, all basic IDL types are directly accessible by special IDLscript identifiers contained in the *CORBA* scope. This *CORBA* scope contains basic CORBA concepts like basic IDL types, standard system exceptions related to CORBA uses, and some other embedded scopes (see Section 3.18, “The Global CORBA Object,” on page 3-40).

3.2.1 IDLscript Representation

Table 3-1 lists the IDLscript identifiers that refer to basic OMG IDL types.

Table 3-1 The IDLscript Representation of OMG IDL Types

Basic OMG IDL Types	IDLscript 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
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

3.2.2 Basic OMG IDL Values

A script can directly manipulate basic IDL types to create basic IDL values as shown in the following example. Operators described in the previous chapter can be used on these values. IDLscript can automatically coerce basic IDL values to basic values when it is necessary as shown on the *v1 + v2 > 100* and *v3 != ""* expressions.

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

3.3 Binding for OMG IDL Module

All IDL modules are directly accessible from the IDLscript interpreter. They are represented by internal objects managed by the Interface Repository cache of the IDLscript interpreter.

3.3.1 OMG IDL Examples

The following example presents some module declarations. The module **GridService** has already been presented in Section 1.4, “An IDLscript Example,” on page 1-6. The module **MA** illustrates the definition of an embedded module **MB**.

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

3.3.2 IDLscript Representation

In IDLscript, access to an IDL module is done simply by providing its IDL module identifier. The evaluation of modules displays the content of the module. This functionality can be used as end-user on-line helping facility. The dotted notation is used to access the contains of a module.

```
>>> GridService
< OMG-IDL module GridService { . . . }; >

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

The previous example illustrates the access to the **`GridService`** and **`MA::MB`** modules. The evaluation of the **`GridService`** module displays its content. Note that as IDL modules are represented by IDLscript objects, they can be assigned to variables (the **`m`** alias).

3.4 Binding for OMG IDL Constant

All IDL constants are directly accessible from the IDLscript interpreter. They are represented by internal objects managed by the Interface Repository cache of the IDLscript interpreter.

3.4.1 OMG IDL Examples

The following example presents some constant declarations: the **PI** and **Math::PI** IDL constants.

```
const double PI = 3.14159;
```

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

3.4.2 IDLscript Representation

In IDLscript, the access to an IDL constant is simply done by providing its IDL constant identifier. This identifier can be prefixed by its IDL module or interface scopes where it is defined. The evaluation of an IDL constant displays the IDL definition of this constant.

```
>>> 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; >
```

The previous example shows how to access the IDL **`PI`** and **`Math::PI`** constants. The evaluation of the **`PI`** constant displays its definition and value. As IDL constants are represented by IDLscript objects, they can be assigned to IDLscript variables to create some kind of aliases as **`c`** and support the **`_type`** attribute.

3.5 Binding for OMG IDL Enum

All IDL enumeration types and values are directly accessible from the IDLscript interpreter. They are represented by internal objects managed by the Interface Repository cache of the IDLscript interpreter.

3.5.1 An OMG IDL Example

Consider the following example which presents an enum declaration. The enumeration **Months** contains all the months of the year.

```
// This definition can be located inside or outside an IDL module or interface
enum Months {
      January, February, March, April, May, June, July, August,
      September, October, November, December
};
```

3.5.2 IDLscript Representation

In IDLscript, the access to an IDL **enum** type is simply done by providing its IDL enumeration identifier. This identifier can be prefixed by its IDL module or interface scopes where it is defined. The evaluation of an IDL **enum** displays the IDL definition of this enumeration.

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

The previous code shows how to access the **Months enum**. This displays all items of this enumeration. As IDL enumeration types are represented by IDLscript objects, they can be assigned to variables to create some kind of aliases.

3.5.3 Enum Values

The creation of an IDL **enum** value needs to specify the selected item belonging to the IDL **enum**. As an IDL **enum** value is represented by an IDLscript object, it is possible to use the typing attributes and methods such as **_type** and **_is_a**.

```
>>> 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
```

For instance, the previous code shows how to create and assign the **January** value of the **Months enum** type to the **a** variable. The last two instructions access to type information managed by the interpreter.

3.6 Binding for OMG IDL Structure

All IDL structure types and values are directly accessible from the IDLscript interpreter. They are represented by internal objects managed by the Interface Repository cache of the IDLscript interpreter.

3.6.1 OMG IDL Examples

Consider the following example which presents some structure declarations. The structure **Point** contains two fields named **x** and **y** with the basic type **double**. The structure **TwoPoints** contains two embedded **Point** structures.

```
// 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;
};
```

3.6.2 IDLscript Representation

In IDLscript, the access to an IDL structure type is simply done by providing its IDL structure identifier. This identifier can be prefixed by its module or interface scopes where it is defined. The evaluation of an IDL structure displays the IDL definition of this structure and all its fields.

```
>>> 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; }; >
```

The previous code presents the access to the **`Point`** and **`TwoPoints`** structures. It is possible to display the entire definition of a structure or only the definition of one field using the dotted notation (**`Point.x`** and **`TwoPoints.a`**). As IDL structure types are represented by IDLscript objects, they can be assigned to variables to create some kind of aliases.

3.6.3 Structure Values

The creation of an IDL structure value is achieved by the calling notation (**`IDLType(field1,...,fieldn)`**). The interpreter checks if the number of given values is equal to the number of the expected IDL fields. If necessary, the interpreter can automatically coerce given values to expected IDL values. For instance, an expected **`long`** field can be initialized by an integer literal. Moreover, a field of an IDL structure type can be initialized by providing an array containing the values of each structure field.

```
>>> 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))
```

The previous code presents some examples of structure value creations. All the fields of the structure must be filled to allow creation and the interpreter coerces integer literals to basic IDL double values. An embedded structure can be defined by several ways: by a literal representation (**`tp1`**), by using variables containing structures already created (**`tp2`**), or by giving the IDL types of the items (**`tp3`**).

3.6.4 Structure Fields

When an IDL structure value is created, the dotted notation allows one to get and set field values. The following example presents some accesses to fields of the previous structure value.

```
>>> 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; } >
```

As IDL structure values are represented by IDLscript objects, it is possible to use common value attributes and methods such as **`_type`** and **`_is_a`**.

3.7 Binding for OMG IDL Union

All IDL union types and values are directly accessible from the IDLscript interpreter. They are represented by internal objects managed by the Interface Repository cache of the IDLscript interpreter.

3.7.1 An OMG IDL Example

Consider the following example which presents a union declaration. In this example, the union named **AnUnion** contains three fields named **m_short**, **m_long**, and **m_float**.

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

3.7.2 IDLscript Representation

In IDLscript, the access to an union type is simply done by providing its IDL union identifier. This identifier can be prefixed by its module or interface scopes where it is defined. The evaluation of an IDL union displays the IDL definition of this union and all its fields.

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

>>> u == AnUnion
true
```

The previous code presents the access to the **AnUnion** union. As IDL union types are represented by IDLscript objects, they can be assigned to variables to create aliases, compared and passed as arguments to procedures.

3.7.3 Union Values

The creation of an IDL union value is achieved by the procedural calling notation **IDLtype(discriminator, value)** and needs two values:

1. the union discriminator value, and
2. the value associated to this discriminator.

The interpreter checks if the discriminator value is correct in relation to the set of case values of the union. Moreover, it checks if the second given value is correct according to the expected union case value. If necessary, the interpreter can automatically coerce the given discriminator and field values to expected IDL values.

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

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

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

The previous code presents some examples of IDL union value creations. As IDL union values are represented by IDLscript objects, it is possible to use common value attributes and methods such as **_type** and **_is_a**.

If there is no field associated with the discriminator value, the union creation is simply done by setting the discriminator value.

```
>>> c= AnUnion(3)
>>> c
AnUnion(3)
```

3.7.4 Union Fields

When an IDL union value is created, the dotted notation allows one to get and set field case values. The special read-only **_d** attribute is provided to access the discriminator value of an IDL union value. When getting a union field, the interpreter checks if the discriminator has the right value and it throws an internal exception to signal that the union does not have the right discriminator. Setting a union field automatically changes the discriminator value. The following example presents some accesses to fields of the previous union value.

```
>>> 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)
```

3.8 Binding for OMG IDL Typedef

All IDL typedef types and values are directly accessible from the IDLscript interpreter. They are represented by internal objects managed by the Interface Repository cache of the IDLscript interpreter.

3.8.1 OMG IDL Examples

Consider the following example which presents an example of typedef declarations. The **Day** typedef refers to the basic **unsigned short** type and the **Coordinate** type refers to the previous **Point** type.

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

3.8.2 IDLscript Representation

In IDLscript, access to an IDL typedef type is done simply by providing its IDL typedef identifier. This identifier can be prefixed by its module or interface scopes where it is defined. The evaluation of an IDL typedef displays the IDL definition of this type definition.

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

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

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

The previous code presents the access to the **Day** and **Coordinate** typedefs. As IDL typedef types are represented by IDLscript objects, they can be assigned to variables to create aliases, compared and passed as arguments to procedures. When an IDL typedef refers to a complex IDL type, it also supports all attributes and methods provided by the aliased type.

3.8.3 Typedef Values

The creation of an IDL typedef value is achieved by the calling notation with a set of initializing values. The number and types of these values must be equal to the number and types needed to create a value of the aliased type.

```
>>> 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
```

The previous code presents some examples of IDL typedef value creations and their uses. The created values support the same attributes and methods as those provided by the aliased type (**`c.x`**). Moreover as IDL typedef values are represented by IDLscript objects, it is possible to use common value attributes and methods such as **`_type`** and **`_is_a`**.

3.9 Binding for OMG IDL Sequence

All IDL sequence types and values are directly accessible from the IDLscript interpreter. They are represented by internal objects managed by the Interface Repository cache of the IDLscript interpreter.

3.9.1 OMG IDL Examples

Consider the following example which presents some sequence declarations: **SeqString** for a **string** sequence, **SeqMonths** for a **Months** sequence, and **SeqPoint** for a **Point** sequence. Only named sequences are supported by IDLscript, no binding for anonymous sequences is provided.

```
// 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;
```

3.9.2 IDLscript Representation

In IDLscript, access to an IDL sequence type is done simply by providing its IDL sequence identifier. This identifier can be prefixed by its module or interface scopes where it is defined. The evaluation of an IDL sequence displays the IDL definition of this type definition.

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

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

>>> s = SeqPoint
>>> s
```

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

The previous code presents the access to the **SeqString**, **SeqMonths**, and **SeqPoint** sequence types. As IDL sequence types are represented by IDLscript objects, they can be assigned to variables to create aliases, compared and passed as arguments to procedures.

3.9.3 Sequence Values

The creation of an IDL sequence value is achieved by the calling notation with a list of values. The type of each value must conform to the item type of the IDL sequence. If necessary, the interpreter automatically coerces given values to required IDL 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; >
```

The previous code presents some examples of IDL sequence value creations. If the list of values is empty, then IDLscript creates an empty sequence value (**SeqMonths()**). The creation of structured value sequences is very simple because each structured value can be provided as an IDLscript array. Then the interpreter checks if the array contains the expected number of values. However it is also possible to use a more typed notation as illustrated by the **s1** creation. As IDL sequence values are represented by IDLscript objects, it is possible to use common value attributes and methods such as **_type** and **_is_a**.

3.9.4 Sequence Items

An IDL sequence value is similar to a basic IDLscript array. It provides the operator `[]` to get and set sequence items, the attribute **length** to obtain the number of items, and can be used in the **for** statement construction. The following example illustrates these functionalities on the previous **SeqPoint** value.

```
>>> 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)
```

3.10 Binding for OMG IDL Array

All IDL array types and values are directly accessible from the IDLscript interpreter. They are represented by internal objects managed by the Interface Repository cache of the IDLscript interpreter.

3.10.1 OMG IDL Examples

Consider the following example which presents some array declarations: **ArrayLong** for a **long** array, and **ArrayPoint** for a **Point** array. IDL arrays have a bounded size defined at declaration. Only named array types are supported by IDLscript, no binding for anonymous arrays is provided.

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

3.10.2 IDLscript Representation

In IDLscript, access to an IDL array type is simply done by providing its IDL array identifier. This identifier can be prefixed by its module or interface scopes where it is defined. The evaluation of an array displays the IDL definition of this type definition.

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

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

The previous code presents access to the **ArrayLong**, and **ArrayPoint** IDL array types. As IDL array types are represented by IDLscript values, they can be assigned to variables to create aliases, compared and passed as arguments to procedures.

3.10.3 Array Values

The creation of an IDL array value is achieved by the calling notation with a list of values. The type of each value must conform to the item type of the IDL array. If necessary, the interpreter automatically coerces given values to required IDL values. Moreover the interpreter checks if the number of given values is equal to the size of the IDL array type.

```
>>> 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(10,10))

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

The previous code presents some examples of IDL array value creations. The creation of structured value IDL arrays is very simple because each structured value can be provided as an IDLscript array. Then the interpreter checks if the array contains the expected number of values. However it is also possible to use a more typed notation as illustrated in Section 3.9.3, “Sequence Values,” on page 3-13. As IDL array values are represented by IDLscript objects, it is possible to use common object attributes and methods such as **`_type`** and **`_is_a`**.

3.10.4 Array Items

An IDL array value is similar to a basic IDLscript array. It provides the operator **`[]`** to get and set array items, the attribute **`length`** to obtain the number of items, and can be used in the **`for`** statement construction. The following example illustrates these functionalities on the previous **`ArrayPoint`** value.

```
>>> 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)
...
```

3.11 Binding for OMG IDL Exception

All IDL exception types and values are directly accessible from the IDLscript interpreter. They are represented by internal objects managed by the IDLscript interpreter.

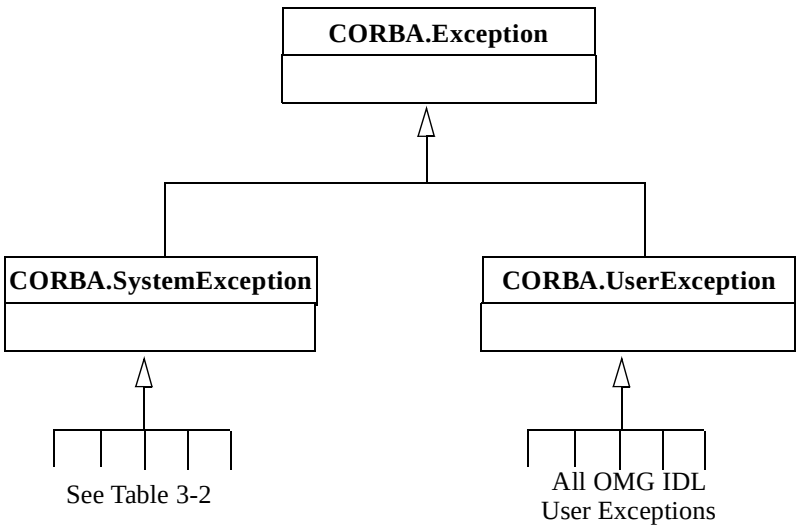

Figure 3-1 The CORBA Exception Type Hierarchy

3.11.1 IDLscript Representation

IDLscript supports all CORBA exception types:

- the System Exceptions representing internal ORB problems, and
- User Exceptions defined in IDL.

Figure 3-1 shows the hierarchy of the IDLscript types representing IDL exception types. All CORBA exception types are transitively subtypes of the CORBA.Exception exception type. This type has two subtypes CORBA.SystemException and CORBA.UserException representing respectively the standard CORBA system exceptions and the IDL user exceptions.

3.11.2 Exception Handling

The CORBA exception types are represented by IDLscript types and are thrown and caught via the exception mechanism presented in Section 2.11, “Exceptions,” on page 2-32. Consider the following example:

```
try {
   # a script code.
   throw CORBA.UNKNOWN()
} catch (CosNaming.NamingContext.AlreadyBound ae) {
   println ("A CosNaming.NamingContext.AlreadyBound exception ",
ae, " has been thrown!")
} catch (CORBA.UserException ue) {
   println ("An IDL exception ", ue, " has been thrown!")
} catch (CORBA.SystemException se) {
   println ("A system  exception ", se, " has been thrown!")
} finally {
   # a finally script code.
}
```

3.11.3 System Exception Types

All standard CORBA system exception types are subtypes of the CORBA.SystemException type. In IDLscript, access to a system exception type is simply done by providing its identifier. This identifier must be prefixed by the *CORBA* scope name like CORBA.INV_OBJREF, CORBA.COMM_FAILURE, or CORBA.OBJECT_NOT_EXIST.

Table 3-2 The IDLscript Identifiers for 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	CORBA.INV_POLICY
CORBA.CODESET_INCOMPATIBLE	

Consider the following examples:

```
>>> 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

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

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

The previous code illustrates access to the CORBA.UNKNOWN exception type. Evaluating an exception type shows the IDL definition of the exception. System exceptions have two fields: the *minor* one and the *completed* one. This latter is a value of the **`CORBA.CompletionStatus`** enumeration type. As system exception types are represented by IDLscript objects, they can be assigned to variables to create aliases, compared and passed as arguments to procedures. Moreover it is possible to use common object attributes and methods such as **`_type`** and **`_is_a`**.

3.11.4 System Exception Values

The creation of a system exception value is achieved by the calling notation **`CORBA.ExceptionName()`**. IDLscript provides three different ways to create these values. The first one needs no parameter and creates a system exception with the **`minor`** field equal to zero and the **`completed`** field equal to the **`COMPLETED_MAYBE`** enumeration value. The second one needs one parameter to initialize the **`minor`** field. The third one takes two parameters to set the **`minor`** and **`completed`** fields.

```
>>> 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

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

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


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

The previous code illustrates the three creation ways of system exceptions. Access to field values is achieved by the dotted notation. Exception values have two fields: the **minor** and **completed** ones. As system exception values are represented by IDLscript objects, it is possible to use common value attributes and methods such as **_type** and **_is_a**. A system exception value is a CORBA.Exception and a CORBA.SystemException as shown in Figure 3-1 on page 3-16.

3.11.5 User Exception Types

Consider the following example that presents some exception declarations. The exception EmptyException contains no field. The exception Exception contains three fields: a simple **string** field, a **Months** enumeration field, and a structured **Point** field.

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

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

In IDLscript, access to an IDL user exception type is done simply by providing its IDL exception identifier. This identifier can be prefixed by its module or interface scopes where it is defined. The evaluation of an IDL exception displays the IDL definition of this exception and all its IDL fields.

```
>>> 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;
}; >

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

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

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

The previous code illustrates the access to the IDL EmptyException and Exception exception types. Evaluating an exception type shows the IDL definition of the exception. As IDL exception types are represented by IDLscript values, they can be assigned to variables to create aliases, compared and passed as arguments to procedures. Moreover it is possible to use common value attributes and methods such as **_type** and **_is_a**. All IDL user exception types are subtypes of the CORBA.Exception and CORBA.UserException types as shown in Figure 3-1 on page 3-16.

3.11.6 User Exception Values

The creation of an IDL exception value is achieved by the calling notation **IDLExceptionType(field1,...,fieldn)**. The interpreter checks if the number of given values is equal to the number of the expected IDL fields. If necessary, the interpreter can automatically coerce given values to expected IDL values. For instance, an expected **string** field can be initialized by a string literal. Moreover, a field of an IDL structure type can be initialized by providing an array containing the value of each structure field.

```
>>> 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
```

The previous code presents some examples of exception value creations. All the fields of the exception must be filled to allow creation and the interpreter coerces literals and arrays to the required IDL values. The dotted notation allows one to get and set field values. As IDL exception values are represented by IDLscript objects, it is possible to use common value attributes and methods such as **_type** and **_is_a**.

3.12 Binding for OMG IDL Interface

All IDL interface types and object references are directly accessible from the IDLscript interpreter. They are represented by internal objects managed by the Interface Repository cache of the IDLscript interpreter.

3.12.1 OMG IDL Examples

Consider the following example which presents some interface declarations. The **Foo** interface contains a **string assignable** attribute, a **double nonassignable** attribute and a **meth** operation. The **AnotherFoo** interface is derived from the **Foo** interface and it adds a new **oper** operation, which illustrates all parameter passing modes. The two operations can raise the EmptyException exception.

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

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

3.12.2 IDLscript Representation

In IDLscript, the access to an IDL interface is simply done by providing its IDL interface identifier. This identifier can be prefixed by its module scopes where it is defined. The evaluation of an IDL interface displays the IDL definition of this type definition with the signature of all attributes and operations and the list of inherited interfaces.

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

>>> a = AnotherFoo
>>> a
< OMG-IDL interface AnotherFoo : Foo {
```

```
    long oper (in long p1, out long p2, inout long p3) raises(Emp-
tyException);
} >

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

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

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

The previous code illustrates access to the **Foo** and **AnotherFoo** interfaces. The evaluation of the **Foo** interface shows the signature of the **assignable** and **nonassignable** attributes and the **meth** operation. The signature of an attribute is composed of its access mode (none or readonly), its type, and its formal name. The signature of an operation is composed of its return type, its formal name, its parameters list (mode, type, and formal name), and its exceptions list.

As IDL interfaces, IDL attributes, IDL operations are represented by IDLscript objects. They can be assigned to variables to create aliases, compared, and passed as arguments to procedures. The hierarchy of IDL interfaces is directly accessible to check interface conformity through the use of common value attributes and methods such as **_type** and **_is_a**.

3.12.3 Object References

Access to CORBA objects requires obtaining related CORBA object references. The creation of these references is simply achieved by the following calling notations **CORBA.Object("StringifiedObjectReference")** or **InterfaceType("Stringified ObjectReference")**. The object reference is given through the following formats: the standard CORBA IOR (i.e., "**IOR:...**") or the Interoperable Name Service URL (i.e., "**iioploc://host:port/object_name**" and "**iiopname//host:port/path_into_a_naming_ context**").

```
>>> objref = CORBA.Object("IOR:.....")
>>> objref._type
< OMG-IDL interface AnotherFoo : Foo {
      long oper (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
```

The first creation notation allows scripts to create an object reference without knowledge about its IDL interface. The second creation notation allows scripts to create an object reference and check if this reference supports a specific IDL interface. However the interpreter only creates the object reference if the given string is correct; otherwise, it raises a CORBA.INV_OBJREF exception. Moreover this object reference is automatically narrowed to the most derived IDL interface type. Then as a result, users can directly and interactively discover the interface supported by the object as shown in the previous example.

As object references are represented by IDLscript objects, they can be assigned to variables, passed as arguments to procedures. Moreover, it is possible to use common object attributes and methods such as **_type** and **_is_a**.

3.12.4 Access to OMG IDL Attributes

Getting and setting IDL attributes is done simply through the dotted notation and by using the IDL identifier of attributes. These accesses are realized by the interpreter *via* the Dynamic Invocation Interface. The interpreter checks the attribute access mode when a script tries to set an attribute (internal IDLscript exception **ReadOnlyAttribute**). If necessary, it also converts automatically the given IDLscript value into the required IDL value. The following example illustrates access to the **assignable** and **nonassignable** 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 ?
```

3.12.5 Invocation of OMG IDL Operations

All IDL operations can be simply invoked with IDLscript using the method calling notation (**object.operation(arg1,...,argn)**). The interpreter automatically checks the number of parameters and coerces given values to IDL values. Invocations are done through the Dynamic Invocation Interface. Exceptions thrown by operations can be easily intercepted thanks to the IDLscript exception mechanism (**try**, **catch**, and **finally** statements).

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

>>> objref.meth(100)
100

>>> try {
```

```
      r = objref.meth(100)
    } catch (EmptyException e) {
      println("The EmptyException has been thrown")
    }

>>> outVariable = Holder()
>>> inoutVariable = Holder(200)
>>> objref.oper (100, outVariable, inoutVariable)
100
>>> outVariable.value
300
```

The previous example illustrates the invocation of the **meth** and **oper** IDL operations. All parameter passing modes are supported by IDLscript. Passing **in** parameters is done by value while **out** and **inout** parameters require using a value of the **Holder** type. As IDLscript is dynamically typed, a Holder can store any IDLscript values (i.e., there is only one **Holder** type). For an **out** parameter, scripts must only create and pass an empty holder to the operation. For an **inout** parameter, scripts must create and pass an initialized holder to the operation. After the invocation, the returned value is available into the holder by its **value** attribute.

3.12.6 Invocation of One-way Operations

Oneway operations are transparently managed by the interpreter. Invocations to operations defined as oneway will always be achieved asynchronously using the same syntactic notation as two-way operations.

3.12.7 Operation Invocation using the Deferred Mode

All IDL operations can be simply invoked using the deferred mode with IDLscript using the method calling notation (**object!operation(arg1,...,argn)**). The interpreter automatically checks the number of parameters and coerces given values to IDL values. Invocations are done through the Dynamic Invocation Interface. Exceptions thrown by operations can be easily intercepted thanks to the IDLscript exception mechanism (**try**, **catch**, and **finally** statements).

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

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

The previous example illustrates deferred invocation of an operation. The result of invocation is obtained using the **value** attribute of the **futureReply** object. Access to the value attribute of the **futureReply** object is blocking while the result is not available.

Inout and **out** parameters are also managed with deferred calls. Consider the following example:

```
>>> outVariable = Holder()
>>> inoutVariable = Holder(200)
>>> futureReply = objref!oper(100,outVariable,inoutVariable)
...
>>> futureReply.value
100
>>> myFutureReplyForMyOutParameter = outVariable.value

>>> myFutureReplyForMyOutParameter.value
300
```

In this example, **outVariable** and **inoutVariable** are **Holder** referencing future objects. Access to the result after invocation is done as for **Holder** in a synchronous invocation (using the **value** attribute). The value contained in the holder is a future object. Access to the real result is done like in the previous example: using the **value** attribute of the **futureReply** object.

If an exception is thrown during the execution of a deferred call, this exception will be thrown in the client side at the first access to a future object involved in this invocation.

Table 3-3 summarizes the functionalities of future objects.

Table 3-3 The Future Object Functionalities

Functionality	Explanation
futureReply.value	Waits for the end of the invocation and returns the result or raises the replied exception if needed.
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 and raises the replied exception if needed.

3.13 Implementing OMG IDL Interfaces

The implementation of IDL interfaces is done simply by IDLscript classes (see Section 2.10, “Classes,” on page 2-28). IDL attributes and operations are implemented by IDLscript instance methods. These instance methods must only follow some naming conventions.

3.13.1 Class Examples

The following example illustrates the implementation of the **Foo** and **AnotherFoo** interfaces presented in Section 3.12.1, “OMG IDL Examples,” on page 3-21. The **Foo** interface is implemented by the **FOO** IDLscript class. The **AnotherFoo** interface is implemented by the **AnotherFOO** IDLscript class. As **AnotherFOO** is a subclass of **FOO**, their instances support instance methods defined in the **FOO** class.

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

3.13.2 OMG IDL Attributes

A class that implements an IDL interface must provide instance methods for IDL attributes. These methods can do any computation on the instance state.

The implementation class must provide a getting method per IDL attribute. The name of these methods is the concatenation of the attribute name and the prefix **_get_** (e.g., **_get_assignable** and **_get_nonassignable**). These methods take one parameter to refer to the current receiver object and must return the (computed) value of the IDL attribute.

For non-readonly IDL attributes, the implementation class must provide a setting method. These methods are named by the IDL attribute name prefixed by **_set_** (e.g., **_set_assignable**). They take two parameters: one to refer to the receiver and another one containing the new value of the IDL attribute. These methods do not return any value.

3.13.3 OMG IDL Operations

Each IDL operation is implemented by an IDLscript method named as the operation, (e.g., **oper** or **meth**).

Implementation methods must take one parameter for the receiver and as many parameters as the IDL operation signature defines. **`In`** parameters are transmitted by value while **`out`** and **`inout`** parameters are received through a **`Holder`** object.

These methods can do any computation on the instance state. They can also throw any CORBA system exception or any user exception defined in the IDL operation signature as shown in the **`oper`** method.

3.13.4 Object Registration

IDLscript provides two different ways to register/unregister object implementations (i.e., IDLscript class instances):

- **The POA approach**: Scripts can use the Portable Object Adapter as defined in the CORBA 2.3 specification. Here, native **`PortableServer::Servant`** and **`PortableServer::Cookie`** are reflected by class instances.
- **A simple connect/disconnect approach**: Here, object implementations are connected/disconnected via the **`connect()`** and **`disconnect()`** methods of the **`CORBA.ORB`** IDLscript object (see Section 3.18.2, “The CORBA::ORB Object,” on page 3-41). Connections may be explicitly or implicitly done by scripts. The disconnection is always explicitly done by scripts. Consider the following 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:000000000000000c49444c3a466f6f3a312e30
00000000010000000000000038000100000000000f3133342e3230362e31
302e3132390000138f0000000000184f422f49442b4e554d0049444c3a46
6f6f3a312e30003200") >
>>> # 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.
```

On the one hand, object implementations may be explicitly connected to the ORB by calling the ORB’s **`connect()`** method. As IDLscript is fully dynamic, this method takes two parameters:

1. the class instance to connect, and

2. the IDL interface which this instance implements.

(Let us note that a third optional parameter can be used to set the ORB-specific object name.)

This way allows scripts to explicitly fix which interfaces an object implements. For example, an IDLscript instance can simultaneously implement several IDL interfaces with different object references.

On the other hand, an object implementation may also be automatically and implicitly connected to the ORB if it is transmitted as a parameter to an IDL operation of a distant CORBA object. This connection is only done if the object implementation was not already connected to an IDL interface which was conformed to the formal parameter type. If the object was already connected to an IDL interface, the previous connection is reused. This approach simplifies the registration of listener objects because registration IDL methods explicitly wait for a specific listener interface. However, this approach can introduce distributed typing problems. For example, if an object implementation is bound to the CosNaming service without explicit connection, then it is implicitly connected to the **`CORBA::Object`** interface.

`PortableServer::POA`, **`PortableServer::current`**, and **`PortableServer Policies`** interfaces must be implemented by the scripting engine. **`PortableServer::POAManager`**, **`PortableServer::AdapterActivator`**, **`PortableServer::ServantManager`**, **`PortableServer::ServantActivator`**, and **`PortableServer::ServantLocator`** are implemented by user classes written in IDLScript.

3.13.5 Object Adapter Run-Time Exceptions

To support IDLscript, an ORB product must provide a reactive or multi-threaded Object Adapter. Then, interactive scripting can be done simultaneously with incoming request handling (i.e., listener callbacks are executed concurrently with interactive scripts). Moreover, some run-time exceptions can be thrown by the IDLscript engine when it receives a CORBA request *via* the Dynamic Skeleton Interface.

Exception	**is thrown when...**
CORBA::BAD_OPERATION	the invoked IDL operation is not supported by the interfaces of the object implementation.

CORBA::OBJ_ADAPTER	the object implementation has been explicitly disconnected from its interfaces.
CORBA::NO_IMPLEMENT	the object implementation class does not provide an implementation for the invoked operation or attribute.
CORBA::BAD_INV_ORDER	the invoked implementation throws an internal exception (i.e., an exception that is not a CORBA exception).

3.14 Binding for OMG IDL Value

All IDL value types and associated values are directly accessible from the IDLscript interpreter. They are reflected by internal objects managed by the IDLscript engine.

3.14.1 OMG IDL Examples

Consider the following examples:

```
valuetype Information sequence<string>;

valuetype Employee  {
   // state definition
     public string name;
     public Information status;
     private unsigned long salary;
  // initializer
     factory init(in string name, in Information status,
                  in unsigned long salary);
  // local operations
     void work();
};
```

This example declares the **Information** boxed value type, a **string** sequence, and the **Employee** value type with public state members (**name** and **status**), a private state member (**salary**), an initializer (**init**), and a local operation (**work**).

3.14.2 IDLscript Representation

In IDLscript, accessing an IDL value type is simply done by providing its IDL value identifier. This identifier can be prefixed by the IDL scope where the value type is defined. Consider following examples:

```
>>> Information
< OMG-IDL valuetype Information sequence<string>; >

>>> Employee
< OMG-IDL valuetype Employee {
```

```
      public string name;
      public Information status;
      private unsigned long salary;
     factory init(in string name, in Information status, in
unsigned long salary);
      void work();
}; >

>>> Employee.name
< OMG-IDL public member string Employee::name; >

>>> Employee.salary
< OMG-IDL private member unsigned long Employee::salary; >

>>> Employee.init
< OMG-IDL factory Employee::init(in string name, in Informa-
tion status, in unsigned long salary); >

>>> w = Employee.work
>>> w
< OMG-IDL operation void Employee::work(); >
```

The evaluation of an IDL value type shows its IDL definition. For example, the boxed type for boxed value types (e.g., **`Information`**) and inheritance, state members, initializers, operations for value types (e.g., **`Employee`**). The evaluation of state members, initializers, and operations shows their signature (as for evaluation of IDL interface attributes and operations). When a boxed value type refers to a complex IDL type, it also supports all attributes and methods provided by the boxed type.

State members, initializers, and operations are reflected by IDLscript objects. They can be assigned to variables in order to create aliases (e.g., **`w`** in the above example), compared and passed as arguments. The inheritance graph composed of IDL value types is directly accessible to check type conformity via the common **`_is_a`** method.

3.14.3 Value Creation

The creation of an IDL boxed value is achieved by the calling notation according to the boxed type. For example, the same number of initialization arguments and each argument must have the expected type. Automatic coercion can also be applied by the IDLscript engine if needed. Consider the following example:

```
>>> i = Information("this", "is", "an", "example")
>>> i
Information("this", "is", "an", "example")
```

The creation of concrete values is achieved by calling one of the initializers declared in the value type. The number and types of arguments must conform to the initializer signature. Again, automatic coercion can be applied if needed. Note that abstract value types cannot be instantiated and concrete value types must declare initializers. Moreover, a local implementation must be known by the IDLscript engine (see

Section 3.15, “Implementing Concrete OMG IDL Values,” on page3-32). If there is no registered local implementation of the concrete value type, then the CORBA::NO_IMPLEMENT exception is thrown. Consider the following example:

```
>>> e = Employee.init("someone", ["info1", "info2"], 0)
>>> e
< Employee value
  name = "someone"
  status = Information("info1", "info2")
>
```

Note that the array passed as second argument to the value initializer is automatically coerced to an **Information** value, and that the evaluation of a value only shows public state members.

As values are reflected by IDLscript objects, they also support the standard **_type** attribute and the **_is_a** method. Consider the following examples:

```
>>> i._type == Information
true

>>> i._is_a(Employee)
false

>>> e._type == Information
false

>>> e._is_a(Employee)
true
```

3.14.4 Null Value

Null values are reflected by the **_null** attribute of the IDLscript reflection of OMG IDL value types. Consider the following example:

```
>>> n = Employee._null

>>> n
< OMG-IDL null Employee value >

>>> n._type == Employee
true
```

3.14.5 Value Manipulation

A boxed value can be manipulated in the same way as an object of the boxed type (i.e., it supports exactly the same scripting notations: operators, attributes, and methods). Consider the following example: the **Information i** value is manipulated as a

string sequence. For example, **i** has a **length** attribute, supports the subscript notation (**[]**) to access and modify items, and can be used in **for** statements (as shown in Section 3.9.4, “Sequence Items,” on page 3-13).

```
>>> i.length
4

>>> i[3] = "example!"
>>> i[3]
"example!"

>>> for s in i print(s, ’ ’)
This is an example!
```

Both concrete and abstract values are manipulated with the dotted notation for invoking local operations. Concrete value operations can only be invoked if the value is associated to a local implementation (see Section 3.15, “Implementing Concrete OMG IDL Values,” on page 3-32); otherwise, a CORBA::NO_IMPLEMENT exception is thrown. Public state members can be get and set by the dotted notation. Private state members are not accessible, an access attempt raises a NotSupported exception. Consider the following examples:

```
>>> e.name = "Mr. Smith"
>>> e.name
"Mr. Smith"

>>> e.status = ["unkwown"]    # automatic coercion
>>> e.status
Information("unkwown")

>>> e.salary
Exception: < NotSupported: < private state member unsigned
long Employee::salary; > >
     File "stdin", line 1 in ?

>>> e.work()
```

Note that if a value without local implementation for its value type is returned by an invocation, then only public state members can be accessed.

3.15 Implementing Concrete OMG IDL Values

The implementation of concrete value types is simply done by IDLscript classes (see Section 2.10, “Classes,” on page 2-28). State members are represented by instance attributes. Initializers and operations are implemented by instance methods.

3.15.1 Example

The following example illustrates the implementation of the **Employee** IDL value type presented in Section 3.14.1, “OMG IDL Examples,” on page3-29. This value type is implemented by the following **EMPLOYEE** class:

```
class EMPLOYEE
{
  proc __EMPLOYEE__(self) { . . . }

  proc init(self, name, status, salary)
  {
    self.name = name
    self.status = status
    self.salary = salary
  }

  proc work(self)  { . . . }
}
```

3.15.2 State Members

Both public and private IDL state members are represented by instance attributes that have the same name. As in IDLscript instance attributes are only defined during their first assignment. It is the programmer’s responsibility to affect each required instance attribute. They must be initialized in the class constructor (e.g., **__EMPLOYEE__** method) and/or in each initializer method.

When a distant operation requires a value type parameter and the user provides an instance, then IDLscript marshals each instance attributes in the same order that it is defined by the value type. If the instance does not have one of the required instance attributes, then the IDLscript engine raises a CORBA::MARSHAL exception.

3.15.3 Initializers

Initializers are instance methods defined in the class implementing a concrete value type. They must have an explicit first parameter that refers to the receiver instance (i.e., as all IDLscript instance methods). Next, parameters will receive arguments passed at the initializer calling time (seeSection 3.14.3, “Value Creation,” on page 3-30). Of course, their number must conform to the initializer signature. The method body must correctly assign instance attributes representing the value state. Other attribute assignments are also allowed, they represent a transient state which is never marshaled on the wire.

The CORBA::NO_IMPLEMENT is thrown if the implementation class does not provide an implementation for an initializer called at concrete value creation time.

3.15.4 Operations

Each IDL value operation is implemented by an instance method named like the operation (e.g., **work**). Operation implementations must take an explicit first parameter referring to the receiver instance, and as many parameters as the operation signature defines. **In** parameters are transmitted by value while **out** and **inout** parameters are received through a **Holder** object.

These methods can do any computation on the instance state. They can also throw any CORBA system exception or any user exception defined in the IDL operation signature.

The CORBA::NO_IMPLEMENT is thrown if the implementation class does not provide an implementation for a called value operation.

3.15.5 Factory Registration

As implementation classes act like value type factories, the **CORBA::ValueFactory** native type is reflected by the IDLscript class concept. These classes must be explicitly registered by the **register_value_factory** method of the **CORBA.ORB** object (see Section 3.18.2, “The CORBA::ORB Object,” on page 3-41). This method takes two parameters: the value type **RepositoryID** and the associated implementation class. This registration allows the IDLscript engine to instantiate classes when users explicitly create a value, or when a value must be unmarshaled from the wire. Consider the following script code:

```
>>> CORBA.ORB.register_value_factory("IDL:Employee:1.0",
EMPLOYEE)
```

As the reflection of an IDL type is also the reflection of the associated **CORBA::TypeCode** (see Section 3.16, “Binding for OMG IDL TypeCode,” on page 3-35), then the **RepositoryID** can directly be obtained from the value type as shown in the following script code.

```
>>> CORBA.ORB.register_value_factory(Employee.id(),EMPLOYEE)
```

Note that if implementation classes are stored into script modules, their registration can be made implicitly by initialization module statements (see Section 2.12.2, “Initialization,” on page 2-37). Then users only need to import these script modules.

3.15.6 Custom Values

Implementation classes of custom marshaled values must explicitly implement marshaling and unmarshaling instance methods. The former named **marshal** takes, as parameters, the receiver instance and a **CORBA::DataOutputStream** object. The latter named **unmarshal** takes the receiver instance and a **CORBA::DataInputStream** object. These stream objects support the standard operations described into the CORBA 2.3 specification and are invoked as abstract values (see Section 3.14.5, “Value Manipulation,” on page 3-31).

Consider the following OMD IDL example:

```
custom valuetype CustomValueExample {
  factory init(in boolean b, in char c, in long l);
};
```

It can be implemented as:

```
class IMPL
{
  proc init(self, b, c, l)
  {
    self.state1 = b
    self.state2 = c
    self.state3 = l
  }

  # dos refers to a CORBA::DataOutputStream object.
  proc marshal(self, dos)
  {
    dos.write_boolean(self.state1)
    dos.write_char(self.state2)
    dos.write_long(self.state3)
  }

  # dis refers to a CORBA::DataInputStream object.
  proc unmarshal(self, dis)
  {
    self.state1 = dis.read_boolean()
    self.state2 = dis.read_char()
    self.state3 = dis.read_long
  }
}
```

3.15.7 Values as Object References

If a concrete value type supports an OMG IDL interface (either concrete or abstract), an instance of the implementation class of this value type can be connected to the ORB (see Section 3.13.4, “Object Registration,” on page 3-27). When this value is passed as parameter to a distant operation call, the IDLscript engine marshals the value state or the IOR according to the standard semantic defined in the CORBA 2.3 specification.

3.16 Binding for OMG IDL TypeCode

As we have seen, the IDLscript language provides a full and transparent binding to any IDL definitions. These IDL types are directly accessible through their related IDL definition name. Then these types can be used anywhere it is needed to provide a CORBA **`TypeCode`** value.

```
>>> ExampleTC
< OMG-IDL interface ExampleTC {
    void send (in TypeCode tc);
}; >

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

>>> tc = CORBA.TypeCode(Foo)
>>> tc
CORBA.TypeCode(Foo)
>>> o.send(tc)
```

The previous code shows how IDL types can be directly sent as CORBA **`TypeCode`** values. The **`ExampleTC`** interface defines the **`send`** operation, which takes a CORBA **`TypeCode`** value as parameter. This operation can be invoked with any IDL type: the basic ones like **`CORBA.Long`**, the user defined ones like **`Point`**, and the interface ones like **`Foo`**. Moreover, **`TypeCode`** values can be explicitly created from the **`CORBA.TypeCode`** binding type.

All the OMG IDL type representations can be managed as IDLscript **`TypeCode`** objects. Table 3-4 enumerates **`TypeCode`** object functionalites.

Table 3-4 The CORBA.TypeCode Functionalities

Functionality	Explanation
`tc.equal(aCorbaType)`	Tests equality between the **`tc TypeCode`** and the **`aCorbaType TypeCode`**.
`tc.equivalent(aCorbaType)`	Tests equivalence between the **`tc TypeCode`** and a **`aCorbaType TypeCode`**.
`tc.get_compact_typecode()`	Returns the compact **`TypeCode`** form of the **`tc TypeCode`**.
`tc.kind()`	Returns the **`CORBA::TCKind`** of the **`tc TypeCode`** and helps to determine what other operations can be invoked on this **`TypeCode`**.
`tc.id()`	Returns the **`CORBA::RepositoryID`** globally identifying the type on the **`TypeCode`**. It can be invoked on object reference, value, structure, union, enumeration, alias, and exception **`TypeCode`**s. Raises a CORBA::TypeCode::BadKind exception if needed.
`tc.name()`	Returns the simple name identifying the type within its enclosing scope. Raises a CORBA::TypeCode::BadKind exception if needed.

Table 3-4 The CORBA.TypeCode Functionalities

`tc.member_count()`	Returns the number of members constituting the type. It can be invoked on structure, union, enumeration, non-boxed value, and exception **`TypeCode`**s. Raises a CORBA::TypeCode::BadKind exception if needed.
`tc.member_name(anIndex)`	Returns the simple name of the member identified by *anIndex*. It can be invoked on structure, union, enumeration, non-boxed value, and exception **`TypeCode`**s. Raises a CORBA::TypeCode::BadKind or a CORBA::TypeCode::Bounds exception if needed.
`tc.member_type(anIndex)`	Returns the **`TypeCode`** describing the type of the member identified by **`anIndex`**. It can be invoked on structure, union, value and exception **`TypeCode`**s. Raises a CORBA::TypeCode::BadKind or a CORBA::TypeCode::Bounds exception if needed.
`tc.member_label(anIndex)`	Returns the label of the union member identified by **`anIndex`**. It can only be invoked on union **`TypeCode`**s. Raises a CORBA::TypeCode::BadKind or a CORBA::TypeCode::Bounds exception if needed.
`tc.discriminator_type()`	Returns the type of all non-default member labels. It can only be invoked on union **`TypeCode`**s. Raises a CORBA::TypeCode::BadKind exception if needed.
`tc.default_index()`	Returns the index of the default member, or -1 if there is no default member. It can only be invoked on union **`TypeCode`**s. Raises a CORBA::TypeCode::BadKind exception if needed.
`tc.length()`	Can be invoked on string, wide string, sequence, and array **`TypeCode`**s. For strings, wide strings, and sequences, it returns the bound, or zero indicating an unbounded string, wide string or sequence. For arrays, it returns the number of elements in the array. Raises a CORBA::TypeCode::BadKind exception if needed.
`tc.content_type()`	Can be invoked on sequence, array, boxed value, and alias **`TypeCode`**s. For sequences and arrays, it returns the element type. For boxed values, it returns the boxed type. For aliases, it returns the original type. Raises a CORBA::TypeCode::BadKind exception if needed.

Table 3-4 The CORBA.TypeCode Functionalities

`tc.fixed_digits()`	Returns the fixed digits of the **`tc`** fixed **`TypeCode`**. Raises a CORBA::TypeCode::BadKind exception if needed.
`tc.fixed_scale()`	Returns the fixed scale of the **`tc`** fixed **`TypeCode`**. Raises a CORBA::TypeCode::BadKind exception if needed.
`tc.member_visibility(anIndex)`	Returns the **`CORBA::Visibility`** of the non-boxed **`tc`** value member identified by **`anIndex`**. Raises a CORBA::TypeCode::BadKind or a CORBA::TypeCode::Bounds exception if needed.
`tc.type_modifier()`	Returns the **`CORBA::ValueModifier`** of the non-boxed **`tc`** valuetype **`TypeCode`**. Raises a CORBA::TypeCode::BadKind exception if needed.
`tc.concrete_base_type()`	Returns the concrete base **`TypeCode`** of the non-boxed **`tc`** valuetype **`TypeCode`**. Raises a CORBA::TypeCode::BadKind exception if needed.

The **`CORBA::TCKind`** enumeration is reflected according to the rules defined in Section 3.5, “Binding for OMG IDL Enum,” on page 3-5.

The **`CORBA::RepositoryID`**, **`CORBA::Visibility`**, and **`CORBA::ValueModifier`** typedef are reflected according to the rules defined in Section 3.8, “Binding for OMG IDL Typedef,” on page 3-11.

The **`CORBA::PRIVATE_MEMBER`**, **`CORBA::PUBLIC_MEMBER`**, **`CORBA::VM_NONE`**, **`CORBA::VM_CUSTOM`**, **`CORBA::VM_ABSTRACT`**, and **`CORBA::VM_TRUNCATABLE`** constants are reflected according to the rules defined in Section 3.4, “Binding for OMG IDL Constant,” on page 3-4.

The **`CORBA::TypeCode::Bounds`** and **`CORBA::TypeCode::BadKind`** user exceptions are reflected according to the rules defined in Section 3.11, “Binding for OMG IDL Exception,” on page 3-16.

3.17 Binding for OMG IDL Any

As we have seen, the IDLscript language allows one to simply create and manipulate any IDL values. These values can be directly created from their related IDL type. Then these values can be used anywhere it is needed to provide a CORBA Any value.

```
>>> ExampleAny
< OMG-IDL interface ExampleAny {
    void send (in any a);
}; >

>>> p = Point(10,10)
>>> foo = Foo("IOR:....")
```

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

>>> 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)
```

The previous example shows how IDL values can be directly sent as CORBA Any values. The **`ExampleAny`** interface defines the **`send`** operation, which takes a CORBA Any value as parameter. This operation can be invoked with any IDL value. The interpreter automatically coerces the IDL value to an Any value like for **`CORBA.Long(10)`**, **`Point(10,10)`**, **`Foo("IOR:....")`** and **`AnotherFoo`** invocations.

Moreover, Any values can be explicitly created from the **`CORBA.Any`** binding type. Such a value supports two attributes:

1. **`type`** to obtain the IDL **`TypeCode`** of the value stored in the Any.
2. **`value`** to obtain the stored value.

Any values used as return values, **`inout`** or **`out`** parameters follow the rules defined in Section 3.12.5, “Invocation of OMG IDL Operations,” on page 3-23.

Some automatic coercions have been defined for the most common types. This feature simplifies the use of IDL specifications using **`CORBA::Any`**. When an any is expected, IDLscript allows scripts to give one of the value of Table 3-5.

Table 3-5 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))

3.18 The Global CORBA Object

The IDLscript engine contains a global object named **CORBA** which is the reflection of the CORBA IDL module. This object defines a scope containing the hierarchy of the previously presented objects: basic IDL types, basic IDL enums, standard CORBA exception types, standard CORBA typedefs and standard CORBA constants. It also contains the **Object** interface and the **ORB** object.

Moreover the **CORBA** object dynamically allows the access to the other IDL definitions contained in the CORBA module if they are populated into the Interface Repository (e.g., **CORBA::Repository**, etc.).

3.18.1 The CORBA::Object Object

The **CORBA.Object** object is the reflection of the base **CORBA::Object** IDL interface. In fact, it is an IDLscript type that defines the standard methods supported by all CORBA object references.

Table 3-6 presents the IDLscript reflection of the **CORBA::Object** operations.

Table 3-6 The Reflection of the CORBA::Object Operations

Object Operation	Reflected by
get_interface	*_get_interface*
is_nil	*_is_nil*
duplicate	Not reflected.
release	Not reflected.
is_a	*_is_a*
non_existent	*_non_existent*
is_equivalent	*_is_equivalent*
hash	*_hash*
create_request	Not reflected.
get_policy	*_get_policy*
get_domain_managers	*_get_domain_managers*
set_policy_overrides	*_set_policy_overrides*

Each **CORBA::Object** operation is reflected by a **CORBA.Object** method prefixed by an underscore ('_') to avoid possible name conflicts with operations defined in IDL interfaces.

As IDLscript provides an automatic garbage collector, the **duplicate** and **release** operations are not reflected in the **CORBA.Object** type.

The **is_a** operation is reflected by the **_is_a** method supported by any IDLscript object. Note that the parameter is not a repository ID **string** but an IDLscript type.

The **create_request** operation is not reflected because the IDLscript language provides a simpler calling notation to invoke object operations (see Section 3.12.5, “Invocation of OMG IDL Operations,” on page 3-23). However, an IDLscript engine must use the DII to invoke distant CORBA object operations.

The **CORBA::InterfaceDef**, **CORBA::Policy**, **CORBA::PolicyType**, **CORBA::PolicyList**, **CORBA::SetOverrideType**, and **CORBA::DomainManagersList** IDL types are reflected in the CORBA global object according to the rules defined in Section 3.12, “Binding for OMG IDL Interface,” on page 3-21, Section 3.8, “Binding for OMG IDL Typedef,” on page 3-11, Section 3.9, “Binding for OMG IDL Sequence,” on page 3-12, and Section 3.5, “Binding for OMG IDL Enum,” on page 3-5.

Following examples illustrate the use of these standard CORBA object operations:

```
>>> o = CORBA.Object("IOR:...")
>>> i = o._get_interface()
>>> i._is_nil()
false
>>> i._is_a(CORBA.InterfaceDef)
true
>>> i._non_existent()
false
>>> o._is_equivalent(i)
false
>>> h = o._hash()
>>> p = o._get_policy( . . . a policy type . . .)
>>> o._set_policy_overrides( . . . a policy list . . .,
                        CORBA.SetOverrideType.ADD_OVERRIDE)
>>> d = o._get_domain_managers()
```

3.18.2 The CORBA::ORB Object

The **CORBA.ORB** object is the reflection of the ORB singleton object and it is initialized at the starting time of an IDLscript engine before its first use. It provides standard ORB operations (i.e., **object_to_string**, **string_to_object**, **list_initial_services**, **resolve_initial_references**, **run**, **shutdown**, etc.). Moreover, it also provides operations to explicitly connect/disconnect a scripting object to/from a CORBA object (see Section 3.13.4, “Object Registration,” on page 3-27). These operations are IDLscript specific but offer a user-friendly way simpler than the POA. However, scripts must use the POA when its advanced features are needed.

The Table 3-7 presents the IDLscript reflection of the **CORBA::ORB** operations.

Table 3-7 The Reflection of the CORBA::ORB Operations

ORB Operation	Reflected by
object_to_string	*object_to_string*
string_to_object	*string_to_object*

Table 3-7 The Reflection of the CORBA::ORB Operations

create_list, *create_operation_list*, *get_default_context*, *send_multiple_requests_oneway*, *send_multiple_requests_deferred*, *poll_next_response*, *get_next_response*	Not reflected.
get_service_information	*get_service_information*
list_initial_services	*list_initial_services*
resolve_initial_references	*resolve_initial_references*
create_*_tc	Not reflected.
work_pending	*work_pending*
perform_work	*perform_work*
run	*run*
shutdown	*shutdown*
destroy	*destroy*
create_policy	*create_policy*
register_value_factory	*register_value_factory*
unregister_value_factory	*unregister_value_factory*
lookup_value_factory	*lookup_value_factory*

The **`CORBA::ORB::InvalidName`**, **`CORBA::ServiceInformation`**, **`CORBA::ServiceOption`**, **`CORBA::ServiceDetail`**, **`CORBA::ServiceDetailType`**, **`CORBA:: ORB::ObjectId`**, **`CORBA::ORB::ObjectIdList`** and other IDL definitions are respectively reflected by IDLscript according to the rules defined in this chapter.

The Dynamic Invocation Interface related operations are not reflected by the IDLscript engine because the language defined herein provides an elegant and user-friendly calling notation to invoke object operations (see Section 3.12.5, “Invocation of OMG IDL Operations,” on page 3-23). However, an IDLscript engine must use the DII to invoke distant CORBA object operations.

The **`TypeCode`** creation operations are not reflected by the IDLscript engine because any OMG IDL definition is automatically available. **`TypeCode`** creations only need to define them using OMG IDL and popularize them into the Interface Repository.

As the **`CORBA::ValueFactory`** native type is reflected by IDLscript classes, the **`register_value_factory`** operation takes a class object as second parameter. Both the **`register_value_factory`** and **`unregister_value_factory`** operations returns the previous registered class object, or the **`Void`** object if none.

The following examples illustrate some of these standard ORB operations:

```
>>> orb = CORBA.ORB

>>> o = orb.object_to_string("IOR: . . .")
>>> orb.object_to_string(o)
"IOR: . . ."

>>> orb.list_initial_services()
CORBA::ORB::ObjectIdList("InterfaceRepository", "NameService
", "RootPOA", ...)
>>> ns = orb.resolve_initial_references("NameService")

>>> orb.work_pending()
true
>>> orb.perform_work()
>>> orb.run()
>>> orb.shutdown(true)

>>> class EMPLOYEE { . . . }
>>> orb.register_value_factory("IDL:Employee:1.0", EMPLOYEE)
>>> vf = orb.lookup_value_factory(Employee.id())
>>> vf == EMPLOYEE
true
>>> vf = orb.unregister_value_factory(Employee.id())
```

Index

Index