The Common Object Request Broker: Architecture and Specification

Digital Equipment Corporation
Hewlett-Packard Company
HyperDesk Corporation
NCR Corporation
Object Design, Inc.
SunSoft, Inc.

OMG Document Number 91.12.1

Revision 1.1

Draft 10 December 1991

Table of Contents

1 Overview 13

1.1 Overview 13

1.2 Typographical Conventions 15

2 The Object Model 17

2.1 Overview 17

2.2 Object Semantics 18

2.2.1 Objects 19
2.2.2 Requests 19
2.2.3 Object Creation and Destruction 20
2.2.4 Types 20
2.2.5 Interfaces 22
2.2.6 Operations 22
2.2.7 Attributes 24

2.3 Object Implementation 24

2.3.1 The Execution Model: Performing Services 24
2.3.2 The Construction Model 25

3 The Common Object Request Broker Architecture 27

3.1 The Structure of an Object Request Broker 28

3.1.1 Object Request Broker 32
3.1.2 Clients 33
3.1.3 Object implementations 33

3.1.4 Object References 34
3.1.5 IDL Interface Definition Language 34
3.1.6 Programming Language Mapping 34
3.1.7 Client Stubs 35
3.1.8 Dynamic Invocation Interface 35
3.1.9 Implementation Skeleton 35
3.1.10 Object Adapters 36
3.1.11 ORB Interface 36
3.1.12 Interface Repository 36
3.1.13 Implementation Repository 37

3.2 Some Example ORBs **37**

3.2.1 Client- and Implementation-resident ORB 37
3.2.2 Server-based ORB 37
3.2.3 System-based ORB 37
3.2.4 Library-based ORB 38

3.3 The Structure of a Client **38**

3.4 The Structure of an Object Implementation **40**

3.5 The Structure of an Object Adapter **41**

3.6 Some example Object Adapters **43**

3.6.1 Basic Object Adapter 43
3.6.2 Library Object Adapter 43
3.6.3 Object-Oriented Database Adapter 43

3.7 The Integration of Foreign Object Systems **44**

4 IDL Syntax and Semantics **45**

4.1 Lexical Conventions **47**

4.1.1 Tokens 50
4.1.2 Comments 50
4.1.3 Identifiers 50
4.1.4 Keywords 51
4.1.5 Literals 52

4.2 Preprocessing **54**

4.3 IDL Grammar **54**

4.4 IDL Specification **58**

4.4.1 Module Declaration 58
4.4.2 Interface Declaration 59

4.5 Inheritance **60**

4.6 Constant Declaration **62**

4.6.1 Syntax 63
4.6.2 Semantics 64

4.7 Type Declaration **65**

4.7.1 Basic Types 66
4.7.2 Constructed Types 68
4.7.3 Template Types 70
4.7.4 Complex Declarator 71

4.8 Exception Declaration **72**

4.9 Operation Declaration **72**

4.9.1 Operation Attribute 73

4.9.2 Parameter Declarations 73
4.9.3 Raises Expressions 74
4.9.4 Context Expressions 74

4.10 Attribute Declaration **75**

4.11 Names and Scoping **76**

4.12 Differences from C++ **78**

4.13 Standard Exceptions **79**

5 C Language Stub Mapping **81**

5.1 Requirements for a Language Mapping **81**

5.1.1 Basic Data Types 82
5.1.2 Constructed Data Types 82
5.1.3 Constants 82
5.1.4 Objects 82
5.1.5 Invocation of Operations 83
5.1.6 Exceptions 83
5.1.7 Attributes 84
5.1.8 ORB Interfaces 84
5.1.9 Language Stub Mapping 85

5.2 Scoped Names **85**

5.3 Mapping for Interfaces **86**

5.4 Inheritance and Operation Names **87**

5.5 Mapping for Attributes **88**

5.6 Mapping for Constants **89**

5.7 Mapping for Basic Data Types **89**

5.8 Mapping for Structure Types **90**

5.9 Mapping for Union Types **90**

5.10 Mapping for Sequence Types **91**

5.11 Mapping for Strings **93**

5.12 Mapping for Arrays **94**

5.13 Mapping for Exception Types **94**

5.14 Implicit Arguments to Operations **95**

5.15 Interpretation of Functions with Empty Argument Lists **95**

5.16 Argument Passing Considerations **95**

5.17 Return Result Passing Considerations **96**

5.18 Summary of Argument/Result Passing **97**

5.19 Handling Exceptions **99**

5.20 Method Routine Signatures **101**

5.21 Include Files **102**

5.22 Pseudo-objects **102**

6 Dynamic Invocation Interface 105

6.1 Overview 105
6.1.1 Common Data Structures 106
6.1.2 Memory Usage 107
6.1.3 Return Statuses and Exceptions 108

6.2 Request Routines 108
6.2.1 create_request 109
6.2.2 add_arg 110
6.2.3 invoke 111
6.2.4 delete 111

6.3 Deferred Synchronous Routines 111
6.3.1 send 111
6.3.2 send_multiple_requests 112
6.3.3 get_response 113
6.3.4 get_next_response 113

6.4 List Routines 113
6.4.1 create_list 114
6.4.2 add_item 114
6.4.3 free 115
6.4.4 free_memory 115
6.4.5 get_count 115
6.4.6 create_operation_list 115

6.5 Context Objects 116

6.6 Context Object Routines 117
6.6.1 get_default_context 118
6.6.2 set_one_value 118
6.6.3 set_values 119
6.6.4 get_values 119
6.6.5 delete_values 120
6.6.6 create_child 120
6.6.7 delete 120

6.7 Native Data Manipulation 121

7 The Interface Repository 123

7.1 Overview 123

7.2 Scope of an Interface Repository 124

7.3 Implementation Dependencies 125

7.4 Basics of the Interface Repository Interface 126
7.4.1 Names 126
7.4.2 Types and TypeCodes 126
7.4.3 Interface Objects 126
7.4.4 Structure and Navigation of Interface Objects 127

7.5 Interface Repository Interfaces 129
7.5.1 Container and Contained Interfaces 129
7.5.2 Repository 132
7.5.3 ModuleDef 133
7.5.4 InterfaceDef 134
7.5.5 AttributeDef 135

7.5.6 OperationDef 135
7.5.7 ParameterDef 136
7.5.8 TypeDef Interface 137
7.5.9 ConstantDef Interface 137
7.5.10 ExceptionDef 138

7.6 TypeCodes **138**

7.6.1 The TypeCode Interface 139
7.6.2 TypeCode Constants 141

8 ORB Interface **143**

8.1 Converting Object References to Strings **143**

8.2 Object Reference Operations **145**

8.2.1 Determining the Object Implementation and Interface 145
8.2.2 Duplicating and Releasing Copies of Object References 145
8.2.3 Nil Object References 146

9 The Basic Object Adapter **147**

9.1 Role of the Basic Object Adapter **147**

9.2 Basic Object Adapter Interface **149**

9.2.1 Registration of Implementations 151
9.2.2 Activation and Deactivation of Implementations 151
9.2.3 Generation and Interpretation of Object References 154
9.2.4 Authentication and Access Control 155
9.2.5 Persistent Storage 156

9.3 C Language Mapping for Object Implementations **156**

9.3.1 Operation-specific details 157
9.3.2 Method signatures 157
9.3.3 Binding Methods to Skeletons 158
9.3.4 BOA and ORB routines 158

10 Interoperability **161**

10.1 The Organization of Multiple ORBs **162**

10.1.1 Reference Embedding 163
10.1.2 Protocol Translation 163
10.1.3 Alternate ORBs 163

11 Glossary **165**

A Standard IDL Types **171**

List of Figures

FIG. 1 Legal Values **21**
FIG. 2 A Request Being Sent Through the Object Request Broker **28**
FIG. 3 The Structure of Object Request Broker Interfaces **29**
FIG. 4 A Client using the Stub or Dynamic Invocation Interface **30**
FIG. 5 An Object Implementation Receiving a Request **31**
FIG. 6 Interface and Implementation Repositories **32**
FIG. 7 The Structure of a Typical Client **39**
FIG. 8 The Structure of a Typical Object Implementation **40**
FIG. 9 The Structure of a Typical Object Adapter **42**
FIG. 10 Different Ways to Integrate Foreign Object Systems **44**
FIG. 11 Legal Multiple Inheritance Example **61**
FIG. 12 Interface Repository Object Containment **128**
FIG. 13 The Structure and Operation of the Basic Object Adapter **149**
FIG. 14 Implementation Activation Policies **153**
FIG. 15 Multiple ORBs **162**

List of Tables

TBL. 1 IDL EBNF Format **46**
TBL. 2 The 114 Alphabetic Characters (Letters) **47**
TBL. 3 Decimal Digits **48**
TBL. 4 The 65 Graphic Characters **48**
TBL. 5 The Formatting Characters **50**
TBL. 6 Keywords **51**
TBL. 7 Punctuation Characters **51**
TBL. 8 Preprocessor Tokens **52**
TBL. 9 Escape Sequences **52**
TBL. 10 Case Label Matching **69**
TBL. 11 Data Type Mappings **89**
TBL. 12 Argument and Result Passing **97**
TBL. 13 Client Argument Storage Responsibilities **98**
TBL. 14 Argument Passing Cases **99**
TBL. 15 C Type Lengths **106**
TBL. 16 Legal TypeCode Kinds and Parameters **139**
TBL. 17 Types Defined by IDL **171**
TBL. 18 Pseudo-objects **172**
TBL. 19 Interface Repository Types **172**
TBL. 20 Minimal Types for Any **173**

1 Overview

1.1 Overview

As defined by the Object Management Group (OMG), the Object Request Broker (ORB) provides the mechanisms by which objects transparently make requests and receive responses. The ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.[1]

1. From *Object Management Architecture Guide*, Revision 1.0, OMG TC Document 90.9.1.

The Common Object Request Broker Architecture and Specification described in this document is a self-contained response to the Request For Proposals (RFP) issued by the ORB Task Force of the OMG, submitted jointly by the following companies:

- Digital Equipment Corporation
- Hewlett-Packard Company
- HyperDesk Corporation
- NCR Corporation
- Object Design, Inc.
- SunSoft, Inc.

It completely replaces the separate proposals previously submitted. At the request of the ORB Task Force, representatives of the above companies worked to find a way to support a common submission that could have rapid commercial availability using technologies each company had already developed. The result is the Common ORB Architecture and Specification, which defines a framework for different ORB implementations to provide common ORB services and interfaces to support portable clients and implementations of objects.

This document is organized as follows:

Chapter 1: Overview
Overview of the document and status.

Chapter 2: The Object Model
The concrete object model for the Common ORB Architecture.

Chapter 3: The Common Object Request Broker Architecture
The overall ORB architecture and interface description.

Chapter 4: IDL Syntax and Semantics
The specification of the Interface Definition Language.

Chapter 5: C Language Stub Mapping
The mapping provided for the C programming language.

Chapter 6: Dynamic Invocation Interface
The dynamic request invocation interface.

Chapter 7: The Interface Repository
The interface (i.e., type) definition repository.

Chapter 8: ORB Interface
The direct interface to the ORB.

Chapter 9: The Basic Object Adapter
The Basic Object Adapter interface.

Chapter 10: Interoperability
A discussion of interoperability between ORBs.

Chapter 11: Glossary — Glossary.

Appendix A: Standard IDL Types
Type definitions available in every ORB.

1.2 Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings, where no distinction is necessary, nor are the type styles used in text where their density would be distracting.

Helvetica bold	IDL language and syntax elements
Times bold	Pseudo-IDL language elements
`Courier bold`	C language elements

Code examples written in pseudo-IDL and C are further identified by means of a comment; unidentified examples are written in IDL.

2 The Object Model

This chapter describes the concrete object model which underlies the Common ORB Architecture. The model is derived from the abstract object model defined by the Object Management Group[1].

2.1 Overview

The object model provides an organized presentation of object concepts and terminology. It defines a partial model for computation that embodies the key characteristics of objects as realized by the submitted technologies.

The OMG object model is *abstract* in that it is not directly realized by any particular technology. The model described here is a *concrete* object model. A concrete object model may differ from the abstract object model in several ways:

1. Object Management Architecture Guide 1.0, Chapter 4, OMG TC Document 90.9.1, Object Management Group, Inc., 492 Old Connecticut Path, Framingham, MA 01701, November 1990.

- it may *elaborate* the abstract object model by making it more specific, for example, by defining the form of request parameters or the language used to specify types
- it may *populate* the model by introducing specific instances of entities defined by the model, for example, specific objects, specific operations, or specific types
- it may *restrict* the model by eliminating entities or placing additional restrictions on their use

An object system is a collection of objects that isolates the requestors of services (clients) from the providers of services by a well-defined encapsulating interface. In particular, clients are isolated from the implementations of services as data representations and executable code.

The object model first describes concepts that are meaningful to clients, including such concepts as object creation and identity, requests and operations, types and signatures. It then describes concepts related to object implementations, including such concepts as methods, execution engines, and activation.

The object model is most specific and prescriptive in defining concepts meaningful to clients. The discussion of object implementation is more suggestive, with the intent of allowing maximal freedom for different object technologies to provide different ways of implementing objects. See Chapter 9 for more information on implementation rules for objects which are managed by the Basic Object Adapter.

There are some other characteristics of object systems that are outside the scope of the object model. Some of these concepts are aspects of application architecture, some are associated with specific domains to which object technology is applied. Such concepts are more properly dealt with in an architectural reference model. Examples of excluded concepts are compound objects, links, copying of objects, change management, and transactions. Also outside the scope of the object model is the model of control and execution.

This object model is an example of a *classical object model*, where a client sends a message to an object. Conceptually, the object interprets the message to decide what service to perform. In the classical model, a message identifies an object and zero or more actual parameters. As in most classical object models, a distinguished first parameter is required, which identifies the operation to be performed; the interpretation of the message by the object involves selecting a method based on the specified operation. Operationally, of course, method selection could be performed either by the object or the ORB.

2.2 Object Semantics

An object system provides services to clients. A *client* of a service is any entity capable of requesting the service.

This section defines the concepts associated with object semantics, i.e. the concepts relevant to clients.

2.2.1 Objects

An object system includes entities known as objects. An *object* is an identifiable, encapsulated entity that provides one or more services that can be requested by a client.

2.2.2 Requests

Clients request services by issuing requests. A *request* is an event, i.e. something that occurs at a particular time. The information associated with a request consists of an operation, a target object, zero or more (actual) parameters, and an optional request context.

A *request form* is a description or pattern that can be evaluated or performed multiple times to cause the issuing of requests. As described in Chapter 4, request forms are defined by particular language bindings. An alternative request form consists of calls to the dynamic invocation interface to create an invocation structure, add arguments to the invocation structure, and to issue the invocation.[2]

A *value* is anything that may be a legitimate (actual) parameter in a request. A value may identify an object, for the purpose of performing the request. A value that identifies an object is called an *object name*. More particularly, a value is an instance of an IDL (Interface Definition Language, defined in Chapter 4) datatype.

An *object reference* is an object name that reliably denotes a particular object. Specifically, an object reference will identify the same object each time the reference is used in a request (subject to certain pragmatic limits of space and time). An object may be denoted by multiple, distinct object references.

A request may have parameters that are used to pass data to the target object; it may also have a request context which provides additional information about the request.

A request causes a service to be performed on behalf of the client. One outcome of performing a service is returning to the client the results, if any, defined for the request.

If an abnormal condition occurs during the performance of a request, an exception is returned. The exception may carry additional return parameters particular to that exception.

2. Descriptions of these request forms may be found in Chapter 5 for the C-language binding for IDL and Chapter 6 for the dynamic invocation interface.

The request parameters are identified by position. A parameter may be an input parameter, an output parameter, or an input-output parameter. A request may also return a single *result value*, as well as any output parameters.

The following semantics hold for all requests:

- any aliasing of parameter values is neither guaranteed removed nor guaranteed preserved
- the order in which aliased output parameters are written is not guaranteed
- any output parameters are undefined if an exception is returned
- the values which may be returned in an input-output parameter may be constrained by the value which was input

Descriptions of the permitted values in requests and the permitted exceptions may be found in §2.2.4 on page 20, and §2.2.6.3 on page 23.

2.2.3 Object Creation and Destruction

Objects can be created and destroyed. From a client's point of view, there is no special mechanism for creating or destroying an object. Objects are created and destroyed as an outcome of issuing requests. The outcome of object creation is revealed to the client in the form of an object reference that denotes the new object.

2.2.4 Types

A *type* is an identifiable entity with an associated predicate (a single-argument mathematical function with a boolean result) defined over values. A value *satisfies* a type if the predicate is true for that value. A value that satisfies a type is called a *member of the type*.

Types are used in signatures to restrict a possible parameter or to characterize a possible result.

The *extension of a type* is the set of values that satisfy the type at any particular time.

An *object type* is a type whose members are objects (literally, values that identify objects). In other words, an object type is satisfied only by (values that identify) objects.

Data types in this model are constrained as follows:

Basic types:

- 16-bit and 32-bit signed and unsigned 2's complement integers
- 32-bit and 64-bit IEEE floating point numbers

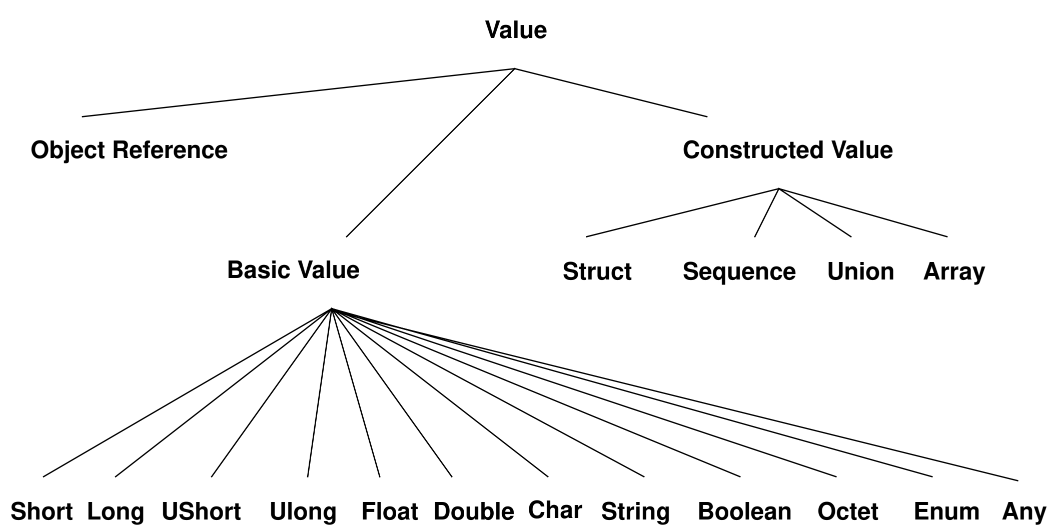
- characters, as defined in ISO Latin-1 (8859.1)
- a boolean type taking the values TRUE and FALSE
- an 8-bit opaque datatype, guaranteed to *not* undergo any conversion during transfer between systems
- enumerated types consisting of ordered sequences of identifiers
- a string type which consists of a variable-length array of characters; the length of the string is available at runtime
- a type “any” which can represent any possible basic or constructed type

Constructed types:

- a record type (called struct), consisting of an ordered set of (name,value) pairs
- a discriminated union type, consisting of a discriminator followed by an instance of a type appropriate to the discriminator value
- a sequence type which consists of a variable-length array of a single type; the length of the sequence is available at runtime
- an array type which consists of a fixed-length array of a single type
- an interface type, which specifies the set of operations which an instance of that type must support

Values in a request are constrained to values which satisfy these type constraints. The legal values are shown in FIG. 1 on page 21. No particular representation for values is defined.

FIG. 1 Legal Values



2.2.5 Interfaces

An *interface* is a description of a set of possible operations that a client may request of an object. An object *satisfies* an interface if it can be specified as the target object in each potential request described by the interface.

An *interface type* is a type that is satisfied by any object (literally, any value that identifies an object) that satisfies a particular interface.

Interfaces are specified in IDL. Interface inheritance provides the composition mechanism for permitting an object to support multiple interfaces. The *principal interface* is simply the most-specific interface that the object supports, and consists of all operations in the transitive closure of the interface inheritance graph.

2.2.6 Operations

An *operation* is an identifiable entity that denotes a service that can be requested.

An operation is identified by an *operation identifier*. An operation is not a value.

An operation has a signature that describes the legitimate values of request parameters and returned results. In particular, a *signature* consists of:

- a specification of the parameters required in requests for that operation
- a specification of the result of the operation
- a specification of the exceptions that may be raised by a request for the operation and the types of the parameters accompanying them
- a specification of additional contextual information that may affect the request
- an indication of the execution semantics the client should expect from a request for the operation

Operations are (potentially) *generic*, meaning that a single operation can be uniformly requested on objects with different implementations, possibly resulting in observably different behavior. Genericity is achieved in this model via interface inheritance in IDL and the total decoupling of implementation from interface specification.

The general form for an operation signature is:

```
[oneway] <op_type_spec> <identifier> (param1, ..., paramL)
        [raises(except1,...,exceptN)] [context(name1, ..., nameM)]
```

where:

- the optional **oneway** keyword indicates that best-effort semantics are expected of requests for this operation; the default semantics are exactly-once if the operation successfully returns results or at-most-once if an exception is returned
- the **<op_type_spec>** is the type of the return result
- the **<identifier>** provides a name for the operation in the interface.
- the operation parameters needed for the operation; they are flagged with the modifiers **in**, **out**, or **inout** to indicate the direction in which the information flows (with respect to the object performing the request)
- the optional **raises** expression indicates which user-defined exceptions can be signalled to terminate a request for this operation; if such an expression is not provided, no user-defined exceptions will be signalled
- the optional **context** expression indicates which request context information will be available to the object implementation; no other contextual information is required to be transported with the request

2.2.6.1 Parameters

A parameter is characterized by its mode and its type. The *mode* indicates whether the value should be passed from client to server (**in**), from server to client (**out**), or both (**inout**). The parameter's type constrains the possible value which may be passed in the direction[s] dictated by the mode.

2.2.6.2 Return Result

The return result is a distinguished **out** parameter.

2.2.6.3 Exceptions

An *exception* is an indication that an operation request was not performed successfully. An exception may be accompanied by additional, exception-specific information.

The additional, exception-specific information is a specialized form of record. As a record, it may consist of any of the types described in §2.2.4 on page 20.

All signatures implicitly include the standard exceptions described in §4.13 on page 79.

2.2.6.4 Contexts

A *request context* provides additional, operation-specific information that may affect the performance of a request.

2.2.6.5 Execution Semantics

Two styles of execution semantics are defined by the object model:

- at-most-once: if an operation request returns successfully, it was performed exactly once; if it returns an exception indication, it was performed at-most-once;
- best-effort: a best-effort operation is a request-only operation, i.e. it cannot return any results and the requester never synchronizes with the completion, if any, of the request.

The execution semantics to be expected is associated with an operation. This prevents a client and object implementation from assuming different execution semantics.

Note that a client is able to invoke an at-most-once operation in a synchronous or deferred-synchronous manner.

2.2.7 Attributes

An interface may have attributes. An attribute is logically equivalent to declaring a pair of accessor functions: one to retrieve the value of the attribute and one to set the value of the attribute.

An attribute may be read-only, in which case only the retrieval accessor function is defined.

2.3 Object Implementation

This section defines the concepts associated with object implementation, i.e. the concepts relevant to realizing the behavior of objects in a computational system.

The implementation of an object system carries out the computational activities needed to effect the behavior of requested services. These activities may include computing the result of the request and updating the system state. In the process, additional requests may be issued.

The implementation model consists of two parts: the execution model and the construction model. The execution model describes how services are performed. The construction model describes how services are defined.

2.3.1 The Execution Model: Performing Services

A requested service is performed in a computational system by executing code that operates upon some data. The data represents a component of the state of the computational system. The code performs the requested service, which may change the state of the system.

Code that is executed to perform a service is called a *method*. A method is an immutable description of a computation that can be interpreted by an execution engine. A method has an immutable attribute called a *method format* that defines the set of execution engines that can interpret the method. An *execution engine* is an abstract machine (not a program) that can interpret methods of certain formats, causing the described computations to be performed. An execution engine defines a dynamic context for the execution of a method. The execution of a method is called a *method activation*.

When a client issues a request, a method of the target object is called. The input parameters passed by the requestor are passed to the method and the output parameters and return value (or exception and its parameters) are passed back to the requestor.

Performing a requested service causes a method to execute that may operate upon an object's persistent state. If the persistent form of the method or state is not accessible to the execution engine, it may be necessary to first copy the method or state into an execution context. This process is called *activation*; the reverse process is called *deactivation*.

2.3.2 The Construction Model

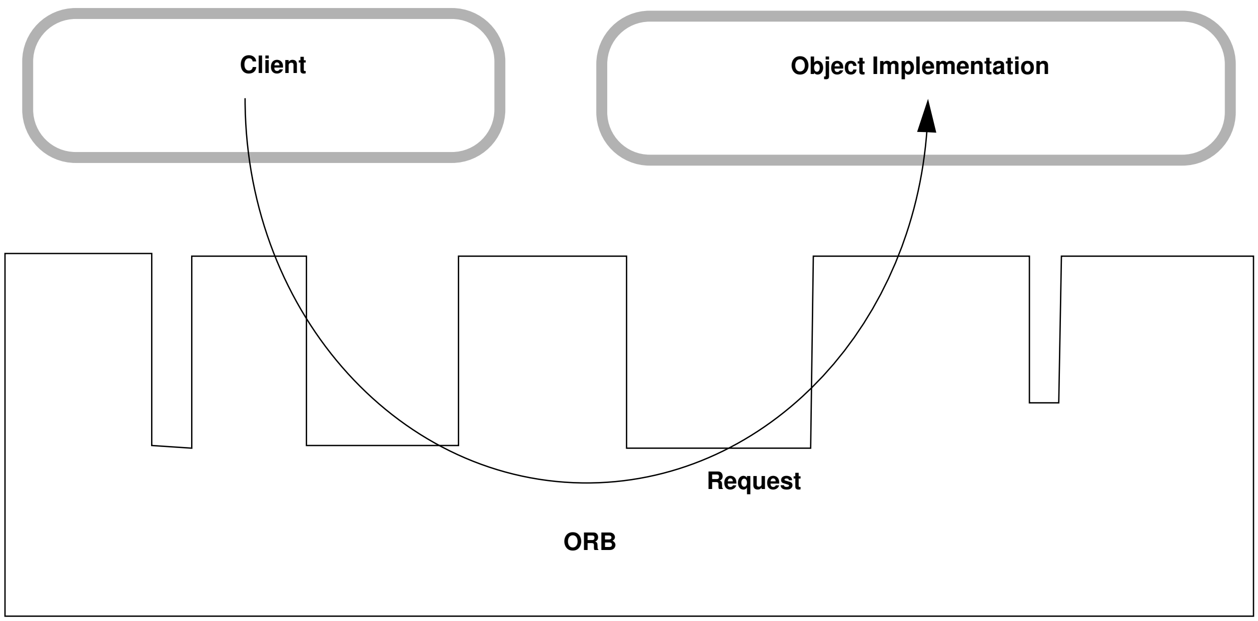
A computational object system must provide mechanisms for realizing behavior of requests. These mechanisms include definitions of object state, definitions of methods, and definitions of how the object infrastructure is to select the methods to execute and to select the relevant portions of object state to be made accessible to the methods. Mechanisms must also be provided to describe the concrete actions associated with object creation, such as association of the new object with appropriate methods.

An *object implementation*—or *implementation*, for short—is a definition that provides the information needed to create an object and to allow the object to participate in providing an appropriate set of services. An implementation typically includes, among other things, definitions of the methods that operate upon the state of an object. It also typically includes information about the intended type of the object.

3 The Common Object Request Broker Architecture

The Common Object Request Broker Architecture (CORBA) is structured to allow integration of a wide variety of object systems. The motivation for some of the features may not be apparent at first, but as we discuss the range of implementations, policies, optimizations, and usages we expect to encompass, the value of the flexibility should become more clear.

FIG. 2 A Request Being Sent Through the Object Request Broker



3.1 The Structure of an Object Request Broker

FIG. 2 on page 28 shows a request being sent by a client to an object implementation.The Client is the entity that wishes to perform an operation on the object and the Object Implementation is the code and data that actually implements the object. The ORB is responsible for all of the mechanisms required to find the object implementation for the request, to prepare the object implementation to receive the request, and to communicate the data making up the request. The interface the client sees is completely independent of where the object is located, what programming language it is implemented in, or any other aspect which is not reflected in the object's interface.

FIG. 3 The Structure of Object Request Broker Interfaces

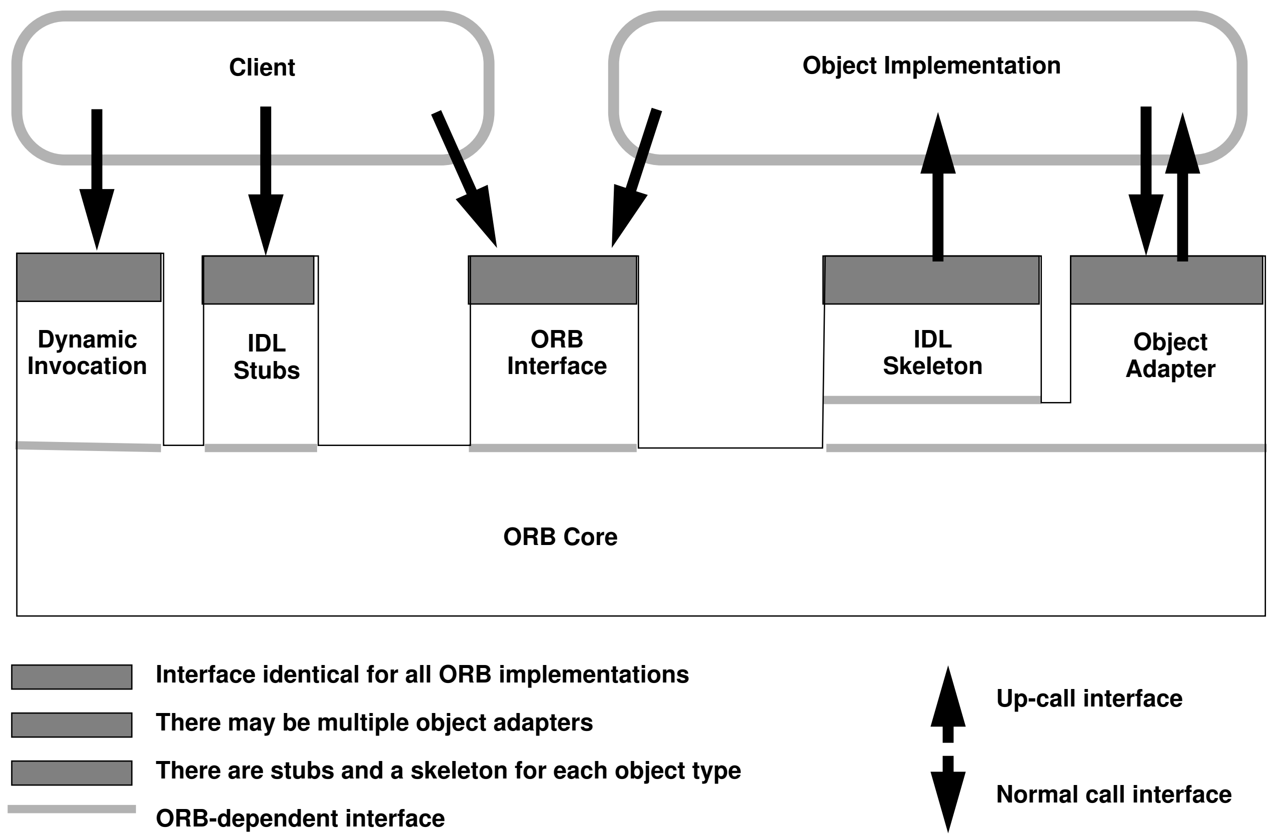


FIG. 3 on page 29 shows the structure of an individual Object Request Broker (ORB). The interfaces to the ORB are shown by striped boxes, and the arrows indicate whether the ORB is called or performs an up-call across the interface.

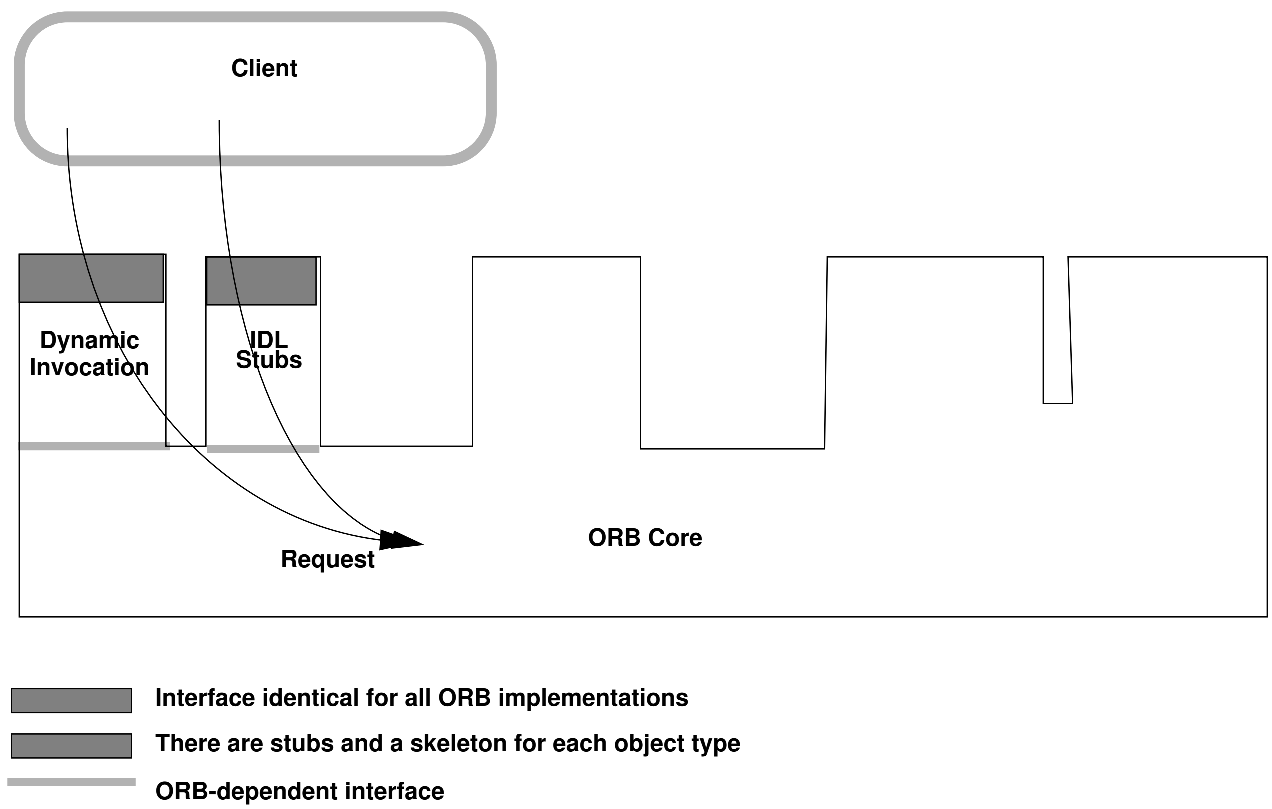
To make a request, the Client can use the Dynamic Invocation interface (the same interface independent of the interface of the target object) or an IDL stub (the specific stub depending on the interface of the target object). The Client can also directly interact with the ORB for some functions.

The Object Implementation receives a request as an up-call through the IDL generated skeleton. The Object Implementation may call the Object Adapter and the ORB while processing a request or at other times.

Definitions of the interfaces to objects can be defined in two ways. Interfaces can be defined statically in an interface definition language, herein called the Interface Definition

Language (IDL). This language defines the types of objects according to the operations that may be performed on them and the parameters to those operations. Alternatively, or in addition, interfaces can be added to an Interface Repository service; this service represents the components of an interface as objects, permitting runtime access to these components. In any ORB implementation, the Interface Definition Language (which may be extended beyond its definition in this document) and the Interface Repository have equivalent expressive power.

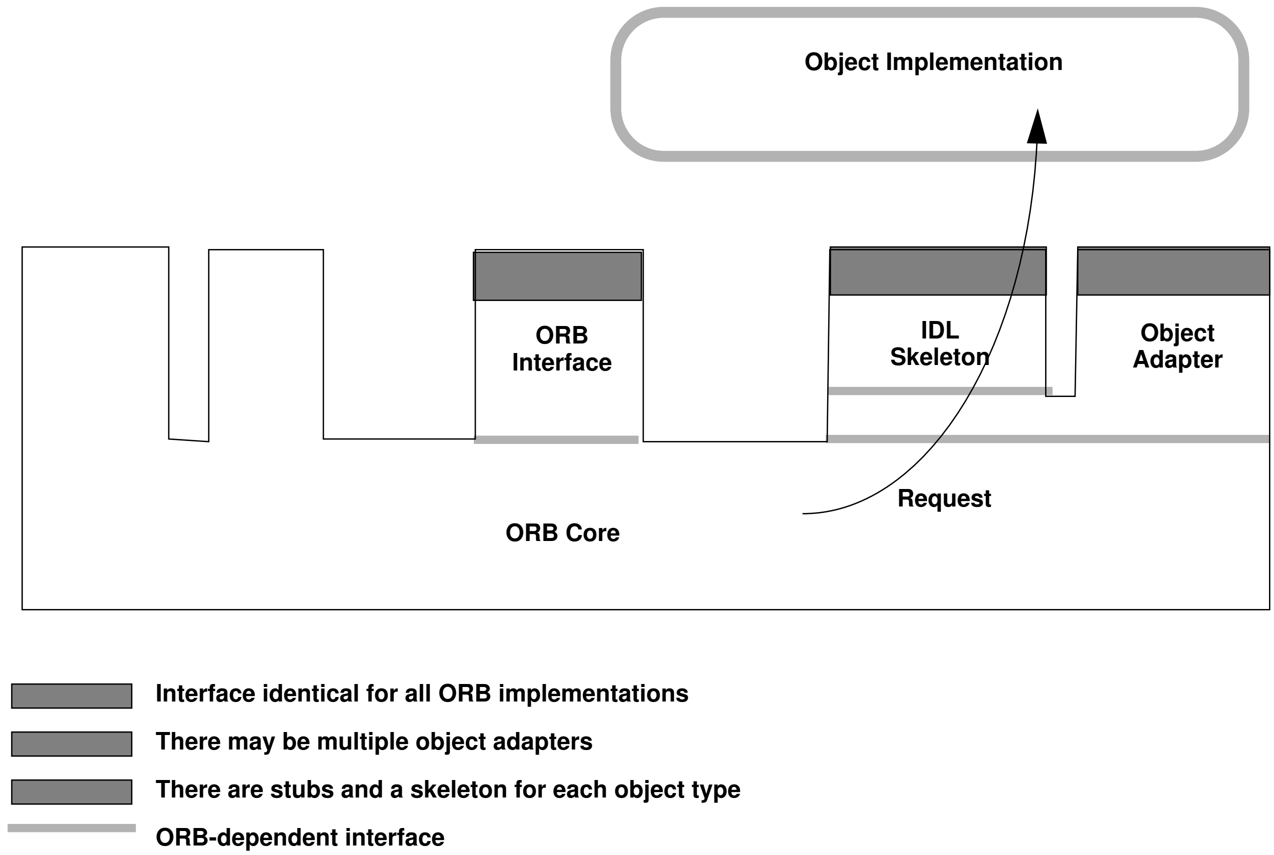
FIG. 4 A Client using the Stub or Dynamic Invocation Interface



The client performs a request by having access to an Object Reference for an object and knowing the type of the object and the desired operation to be performed. The client initiates the request by calling stub routines that are specific to the object or by constructing the request dynamically (see FIG. 4 on page 30).

The dynamic and stub interface for invoking a request satisfy the same request semantics, and the receiver of the message cannot tell how the request was invoked.

FIG. 5 An Object Implementation Receiving a Request



The ORB locates the appropriate implementation code, transmits parameters and transfers control to the Object Implementation through an IDL skeleton (see FIG. 5 on page 31). Skeletons are specific to the interface and the object adapter. In performing the request, the object implementation may obtain some services from the ORB through the Object Adapter. When the request is complete, control and output values are returned to the client.

The Object Implementation may choose which Object Adapter to use. This decision is based on what kind of services the Object Implementation requires.

FIG. 6 Interface and Implementation Repositories

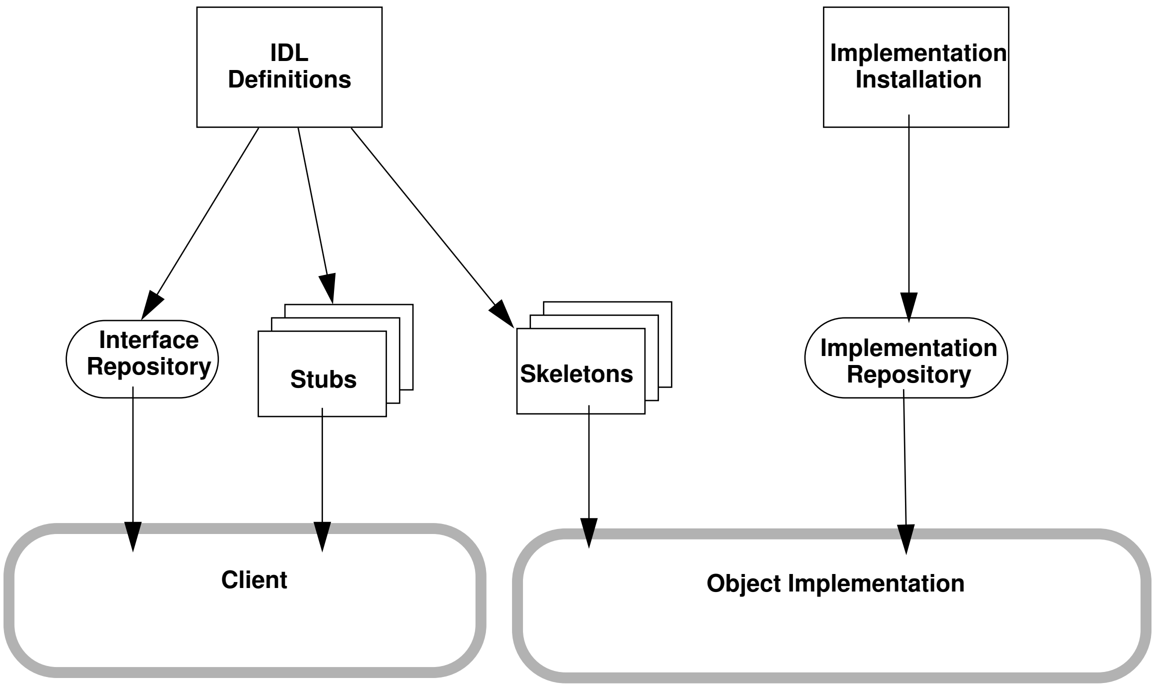


FIG. 6 on page 32 shows how interface and implementation information is made available to clients and object implementations. The interface is defined in IDL and/or in the Interface Repository; the definition is used to generate the client Stubs and the object implementation Skeletons.

The Object Implementation information is provided at installation time and is stored in the Implementation Repository for use during request delivery.

3.1.1 Object Request Broker

In the architecture, the ORB is not required to be implemented as a single component, but rather it is defined by its interfaces. Any ORB implementation that provides the appropriate interface is acceptable. The interface is organized into three categories:

1. those operations that are the same for all ORB implementations,
2. those operations that are specific to particular types of objects, and
3. those operations that are specific to particular styles of object implementations.

Different ORBs may make quite different implementation choices, and, together with the IDL compilers, repositories, and various Object Adapters, provide a set of services to clients and implementations of objects that have different properties and qualities.

There may be multiple ORB implementations (also described as multiple ORBs) which have different representations for object references and different means of performing invocations. It may be possible for a client to simultaneously have access to two object references managed by different ORB implementations. When two ORBs are intended to work together, those ORBs must be able to distinguish their object references. It is not the responsibility of the client to do so.

The ORB Core is that part of the ORB that provides the basic representation of objects and communication of requests. CORBA is designed to support different object mechanisms, and it does so by structuring the ORB with components above the ORB Core, which provide interfaces that can mask the differences between ORB Cores.

3.1.2 Clients

A client of an object has access to an object reference for the object, and invokes operations on the object. A client knows only the logical structure of the object according to its interface and experiences the behavior of the object through invocations. Although we will generally consider a client to be a program or process initiating requests on an object, it is important to recognize that something is a client relative to a particular object. For example, the implementation of one object may be a client of other objects

Clients generally see objects and ORB interfaces through the perspective of a language mapping, bringing the ORB right up to the programmer's level. Clients are maximally portable and should be able to work without source changes on any ORB that supports the desired language mapping with any object instance that implements the desired interface. Clients have no knowledge of the implementation of the object, which object adapter is used by the implementation, or which ORB is used to access it.

3.1.3 Object implementations

An object implementation provides the semantics of the object, usually by defining data for the object instance and code for the object's methods. Often the implementation will use other objects or additional software to implement the behavior of the object. In some cases, the primary function of the object is to have side-effects on other things that are not objects.

A variety of object implementations can be supported, including separate servers, libraries, a program per method, an encapsulated application, an object-oriented database, etc. Through the use of additional object adapters, it is possible to support virtually any style of object implementation.

Generally, object implementations do not depend on the ORB or how the client invokes the object. Object implementations may select interfaces to ORB-dependent services by the choice of Object Adapter. Object implementations are portable across any ORB that supports the desired language mapping and implements the desired Object Adapter.

3.1.4 Object References

An Object Reference is the information needed to specify an object within an ORB. Both clients and object implementations have an opaque notion of object references according to the language mapping, and thus are insulated from the actual representation of them. Two ORB implementations may differ in their choice of Object Reference representations.

The representation of an object reference handed to a client is only valid for the lifetime of that client.

All ORBs must provide the same language mapping to an object reference (usually referred to as an Object) for a particular programming language. This permits a program written in a particular language to access object references independent of the particular ORB. The language mapping may also provide additional ways to access object references in a typed way for the convenience of the programmer.

There is a distinguished object reference, guaranteed to be different from all object references, that denotes no object.

3.1.5 IDL Interface Definition Language

The IDL Interface Definition Language defines the types of objects by specifying their interfaces. An interface consists of a set of named operations and the parameters to those operations. Note that although IDL provides the conceptual framework for describing the objects manipulated by the ORB, it is not necessary for there to be IDL source code available for the ORB to work. As long as the equivalent information is available in the form of stub routines or a runtime interface repository, a particular ORB may be able to function correctly.

IDL is the means by which a particular object implementation tells its potential clients what operations are available and how they should be invoked. From the IDL definitions, it is possible to map CORBA objects into particular programming languages or object systems.

3.1.6 Programming Language Mapping

Different object-oriented or non-object-oriented programming languages may prefer to access CORBA objects in different ways. For object-oriented languages, it may be desirable to see CORBA objects as programming language objects. Even for non-object-oriented languages, it is a good idea to hide the exact ORB representation of the object reference, method names, etc. A particular language mapping to CORBA should be the same for all ORB implementations; however, language mappings for languages other than C are not currently part of the specification. The language mapping includes definition of the language-specific data types and procedure interfaces to access objects through the ORB. It includes the structure of the client stub interface, the dynamic invocation interface, the implementation skeleton, the object adapters, and the direct ORB interface.

The language mapping also defines the interaction between object invocations and the threads of control in the client or implementation. The most common mappings provide synchronous calls, in that the routine returns when the object operation completes. Additional mappings may be provided to allow a call to be initiated and control returned to the program. In such cases, additional language-specific routines must be provided to synchronize the program's threads of control with the object invocation.

3.1.7 Client Stubs

For a particular language mapping, there will be a programming interface to the stubs for each interface type. Generally, the stubs will present access to the IDL-defined operations on an object in a way that is easy for programmers to predict once they are familiar with IDL and the language mapping for the particular programming language. The stubs make calls on the rest of the ORB using interfaces that are private to, and presumably optimized for, the particular ORB Core. If more than one ORB is available, there may be different stubs corresponding to the different ORBs. In this case, it is necessary for the ORB and language mapping to cooperate to associate the correct stubs with the particular object reference.

3.1.8 Dynamic Invocation Interface

An interface is also available that allows the dynamic construction of object invocations, that is, rather than calling a stub routine that is specific to a particular operation on a particular object, a client may specify the object to be invoked, the operation to be performed, and the set of parameters for the operation through a call or sequence of calls. The client code must supply information about the operation to be performed and the types of the parameters being passed (perhaps obtaining it from an Interface Repository or other runtime source). The nature of the dynamic invocation interface may vary substantially from one programming language mapping to another.

3.1.9 Implementation Skeleton

For a particular language mapping, and possibly depending on the object adapter, there will be an interface to the methods that implement each type of object. The interface will generally be an up-call interface, in that the object implementation writes routines that conform to the interface and the ORB calls them through the skeleton.

The existence of a skeleton does not imply the existence of a corresponding client stub (clients can also make requests via the dynamic invocation interface).

It is possible to write an object adapter that does not use skeletons to invoke implementation methods. For example, it may be possible to create implementations dynamically for languages such as Smalltalk.

3.1.10 Object Adapters

An object adapter is the primary way that an object implementation accesses services provided by the ORB. There are expected to be a few object adapters that will be widely available, with interfaces that are appropriate for specific kinds of objects. Services provided by the ORB through an Object Adapter often include: generation and interpretation of object references, method invocation, security of interactions, object and implementation activation and deactivation, mapping object references to implementations, and registration of implementations.

The wide range of object granularities, lifetimes, policies, implementation styles, and other properties make it difficult for the ORB Core to provide a single interface that is convenient and efficient for all objects. Thus, through Object Adapters, it is possible for the ORB to target particular groups of object implementations that have similar requirements with interfaces tailored to them.

3.1.11 ORB Interface

The ORB Interface is the interface that goes directly to the ORB which is the same for all ORBs and does not depend on the object's interface or object adapter. Because most of the functionality of the ORB is provided through the object adapter, stubs, skeleton, or dynamic invocation, there are only a few operations that are common across all objects. These operations are useful to both clients and implementations of objects.

3.1.12 Interface Repository

The Interface Repository is a service that provides persistent objects that represent the IDL information in a form available at runtime. The Interface Repository information may be used by the ORB to perform requests. Moreover, using the information in the Interface Repository, it is possible for a program to encounter an object whose interface was not known when the program was compiled, yet, be able to determine what operations are valid on the object and make an invocation on it.

In addition to its role in the functioning of the ORB, the Interface Repository is a common place to store additional information associated with interfaces to ORB objects. For example, debugging information, libraries of stubs or skeletons, routines that can format or browse particular kinds of objects, etc., might be associated with the Interface Repository.

3.1.13 Implementation Repository

The Implementation Repository contains information that allows the ORB to locate and activate implementations of objects. Although most of the information in the Implementation Repository is specific to an ORB or operating environment, the Implementation Repository is the conventional place for recording such information. Ordinarily, installation of implementations and control of policies related to the activation and execution of object implementations is done through operations on the Implementation Repository.

In addition to its role in the functioning of the ORB, the Implementation Repository is a common place to store additional information associated with implementations of ORB objects. For example, debugging information, administrative control, resource allocation, security, etc., might be associated with the Implementation Repository.

3.2 Some Example ORBs

There are a wide variety of ORB implementations possible within the Common ORB Architecture. This section will illustrate some of the different options. Note that a particular ORB might support multiple options and protocols for communication.

3.2.1 Client- and Implementation-resident ORB

If there is a suitable communication mechanism present, an ORB can be implemented in routines resident in the clients and implementations. The stubs in the client either use a location-transparent IPC mechanism or directly access a location service to establish communication with the implementations. Code linked with the implementation is responsible for setting up appropriate databases for use by clients.

3.2.2 Server-based ORB

To centralize the management of the ORB, all clients and implementations can communicate with one or more servers whose job it is to route requests from clients to implementations. The ORB could be a normal program as far as the underlying operating system is concerned, and normal IPC could be used to communicate with the ORB.

3.2.3 System-based ORB

To enhance security, robustness, and performance, the ORB could be provided as a basic service of the underlying operating system. Object references could be made unforgeable, reducing the expense of authentication on each request. Because the operating system could know the location and structure of clients and implementations, it would be possible for a variety of optimizations to be implemented, for example, avoiding marshalling when both are on the same machine.

3.2.4 Library-based ORB

For objects that are light-weight and whose implementations can be shared, the implementation might actually be in a library. In this case, the stubs could be the actual methods. This assumes that it is possible for a client program to get access to the data for the objects and that the implementation trusts the client not to damage the data.

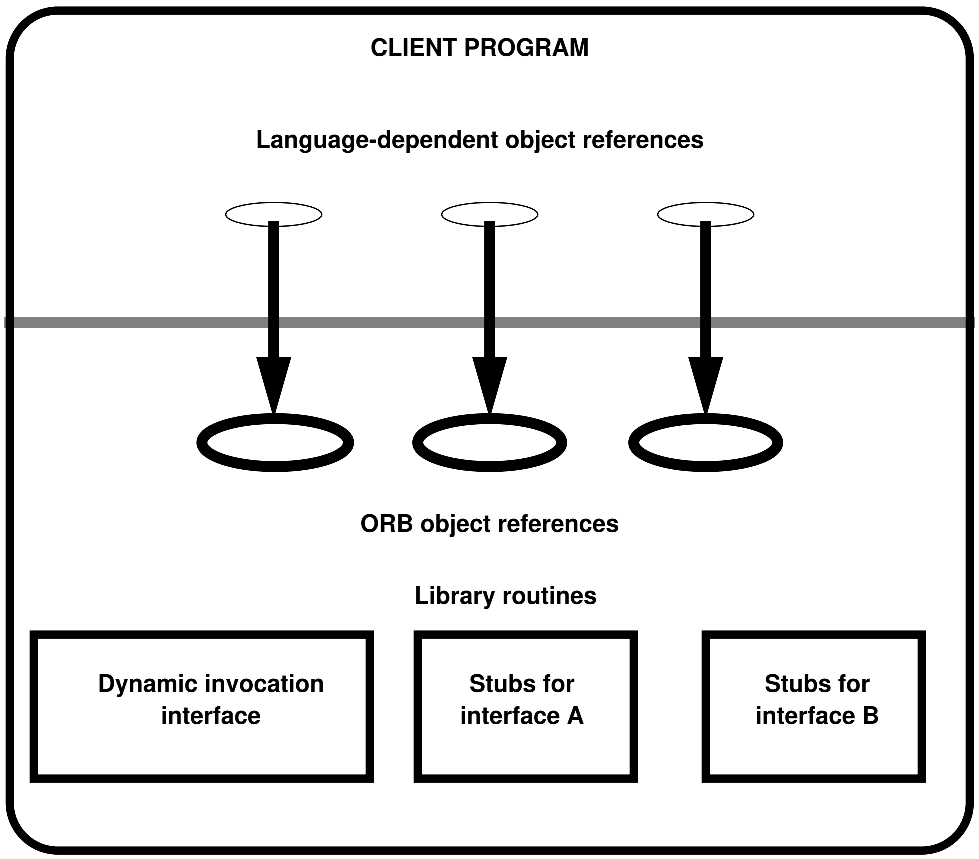
3.3 The Structure of a Client

A client of an object has an object reference that refers to that object. An object reference is a token that may be invoked or passed as a parameter to an invocation on a different object. Invocation of an object involves specifying the object to be invoked, the operation to be performed, and parameters to be given to the operation or returned from it.

The ORB manages the control transfer and data transfer to the object implementation and back to the client. In the event that the ORB cannot complete the invocation, an exception response is provided. Ordinarily, a client calls a routine in its program that performs the invocation and returns when the operation is complete.

Clients access object-type-specific stubs as library routines in their program (see FIG. 7 on page 39). The client program thus sees routines callable in the normal way in its programming language. All implementations will provide a language-specific data type to use to refer to objects, often an opaque pointer. The client then passes that object reference to the stub routines to initiate an invocation. The stubs have access to the object reference representation and interact with the ORB to perform the invocation. (See §5.1 on page 81 for more details on language mapping of object references.)

FIG. 7 The Structure of a Typical Client



An alternative set of library code is available to perform invocations on objects, for example when the object was not defined at compile time. In that case, the client program provides additional information to name the type of the object and the method being invoked, and performs a sequence of calls to specify the parameters and initiate the invocation.

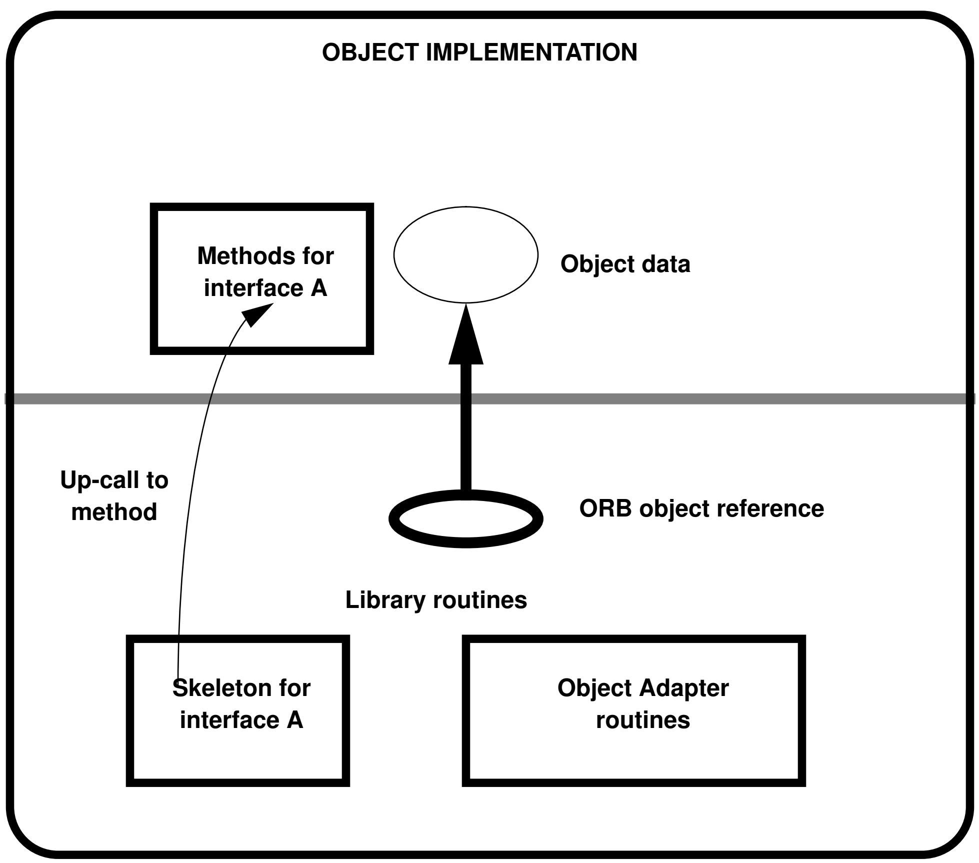
Clients most commonly obtain object references by receiving them as output parameters from invocations on other objects for which they have references. When a client is also an implementation, it receives object references as input parameters on invocations to objects it implements. An object reference can also be converted to a string that can be stored in files or preserved or communicated by different means and subsequently turned back into an object reference by the ORB that produced the string.

3.4 The Structure of an Object Implementation

An Object Implementation provides the actual state and behavior of an object. The object implementation can be structured in a variety of ways. Besides defining the methods for the operations themselves, an implementation will usually define procedures for activating and deactivating objects and will use other objects or non-object facilities to make the object state persistent, to control access to the object, as well as to implement the methods.

The object implementation (see FIG. 8 on page 40) interacts with the ORB in a variety of ways to establish its identity, to create new objects, and to obtain ORB-dependent services. It primarily does this via access to an Object Adapter, which provides an interface to ORB services that is convenient for a particular style of object implementation.

FIG. 8 The Structure of a Typical Object Implementation



Because of the range of possible object implementations, it is difficult to be definitive about how in general an object implementation is structured. See Chapter 9 for the structure of object implementations that use the Basic Object Adapter.

When an invocation occurs, the ORB Core, object adapter, and skeleton arrange that a call is made to the appropriate method of the implementation. A parameter to that method specifies the object being invoked, which the method can use to locate the data for the object. Additional parameters are supplied according to the skeleton definition. When the method is complete, it returns, causing output parameters or exception results to be transmitted back to the client.

When a new object is created, the ORB may be notified so that the it knows where to find the implementation for that object. Usually, the implementation also registers itself as implementing objects of a particular interface, and specifies how to start up the implementation if it is not already running.

Most object implementations provide their behavior using facilities in addition to the ORB and object adapter. For example, although the Basic Object Adapter provides some persistent data associated with an object, that relatively small amount of data is typically used as an identifier for the actual object data stored in a storage service of the object implementation's choosing. With this structure, it is not only possible for different object implementations to use the same storage service, it is also possible for objects to choose the service that is most appropriate for them.

3.5 The Structure of an Object Adapter

An object adapter (see FIG. 9 on page 42) is the primary means for an object implementation to access ORB services such as object reference generation. An object adapter exports a public interface to the object implementation, and a private interface to the skeleton. It is built on a private ORB-dependent interface.

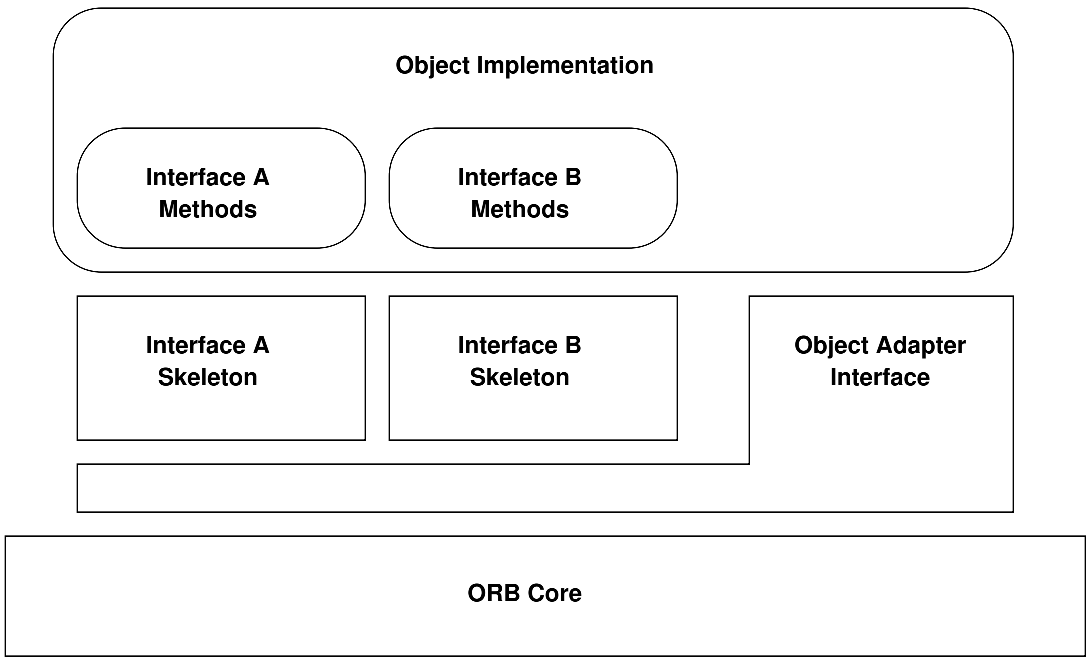
Object adapters are responsible for the following functions:

- generation and interpretation of object references
- method invocation
- security of interactions
- object and implementation activation and deactivation
- mapping object references to the corresponding object implementations
- registration of implementations

These functions are performed using the ORB Core and any additional components necessary. Often, an object adapter will maintain its own state to accomplish its tasks. It may be possible for a particular object adapter to delegate one or more of its responsibilities to the Core upon which it is constructed.

As shown in FIG. 9 on page 42, the Object Adapter is implicitly involved in invocation of the methods, although the direct interface is through the skeletons. For example, the Object Adapter may be involved in activating the implementation or authenticating the request.

FIG. 9 The Structure of a Typical Object Adapter



The Object Adapter defines most of the services from the ORB that the Object Implementation can depend on. Different ORBs will provide different levels of service and different operating environments may provide some properties implicitly and require others to be added by the Object Adapter. For example, it is common for Object Implementations to want to store certain values in the object reference for easy identification of the object on an invocation. If the Object Adapter allows the implementation to specify such values when a new object is created, it may be able to store them in the object reference for those ORBs that permit it. If the ORB Core does not provide this feature, the Object Adapter would record the value in its own storage and provide it to the implementation on an invocation. With Object Adapters, it is possible for an Object Implementation to have access to a service whether or not it is implemented in the ORB Core—if the ORB Core provides it, the adapter simply provides an interface to it; if not, the adapter must implement it on top of the ORB Core. Every instance of a particular adapter must provide the same interface and service for all the ORBs it is implemented on.

It is also not necessary for all Object Adapters to provide the same interface or functionality. Some Object Implementations have special requirements, for example, an object-oriented database system may wish to implicitly register its many thousands of objects without doing individual calls to the Object Adapter. In such a case, it would be impractical and unnecessary for the object adapter to maintain any per-object state. By using an object adapter interface that is tuned towards such object implementations, it is possible to take advantage of particular ORB Core details to provide the most effective access to the ORB.

3.6 Some example Object Adapters

There are a variety of possible object adapters. However, since the object adapter interface is something that object implementations depend on, it is desirable that there be as few as practical. Most object adapters are designed to cover a range of object implementations, so only when an implementation requires radically different services or interfaces should a new object adapter be considered. In this section, we describe three object adapters that might be useful.

3.6.1 Basic Object Adapter

This specification defines an object adapter that can be used for most ORB objects with conventional implementations (See Chapter 9). For this object adapter, implementations are generally separate programs. It allows there to be a program started per method, a separate program for each object, or a shared program for all instances of the object type. It provides a small amount of persistent storage for each object, which can be used as a name or identifier for other storage, for access control lists, or other object properties. If the implementation is not active when an invocation is performed, the BOA will start one.

3.6.2 Library Object Adapter

This object adapter is primarily used for objects that have library implementations. It accesses persistent storage in files, and does not support activation or authentication, since the objects are assumed to be in the clients program.

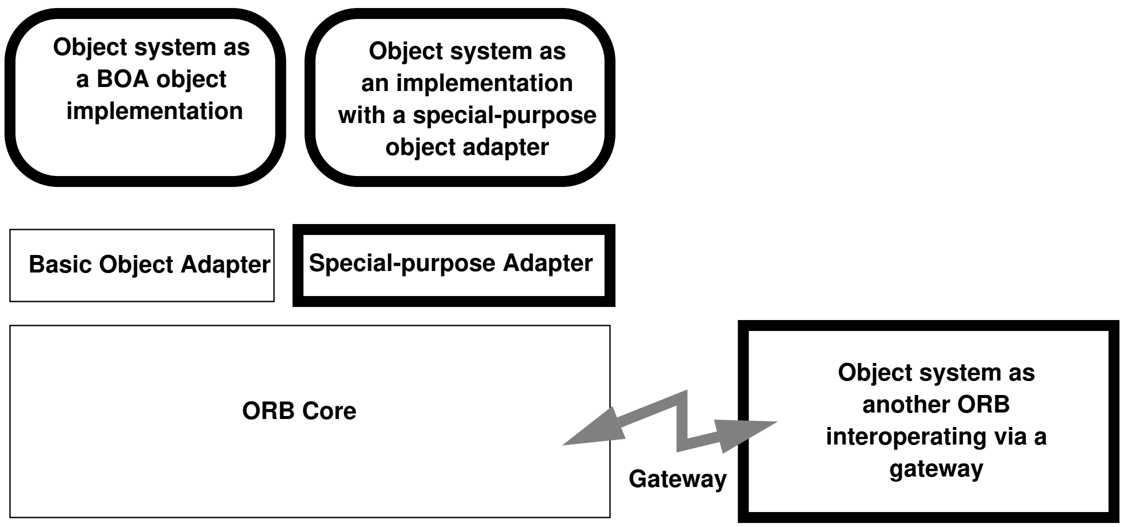
3.6.3 Object-Oriented Database Adapter

This adapter uses a connection to an object-oriented database to provide access to the objects stored in it. Since the OODB provides the methods and persistent storage, objects may be registered implicitly and no state is required in the object adapter.

3.7 The Integration of Foreign Object Systems

The Common ORB Architecture is designed to allow interoperation with a wide range of object systems (see FIG. 10 on page 44). Because there are many existing object systems, a common desire will be to allow the objects in those systems to be accessible via the ORB. For those object systems that are ORBs themselves, they may be connected to other ORBs through the mechanisms described in chapter 10 on page 161.

FIG. 10 Different Ways to Integrate Foreign Object Systems



For object systems that simply want to map their objects into ORB objects and receive invocations through the ORB, one approach is to have those object systems appear to be implementations of the corresponding ORB objects. The object system would register its objects with the ORB and handle incoming requests, and could act like a client and perform outgoing requests.

In some cases, it will be impractical for another object system to act like a BOA object implementation. An object adapter could be designed for objects that are created in conjunction with the ORB and that are primarily invoked through the ORB. Another object system may wish to create objects without consulting the ORB, and might expect most invocations to occur within itself rather than through the ORB. In such a case, a more appropriate object adapter might allow objects to be implicitly registered when they are passed through the ORB.

4 IDL Syntax and Semantics

IDL (the Interface Definition Language) is the language used to describe the interfaces that client objects call and object implementations provide. An interface definition written in IDL completely defines the interface and fully specifies each operation's parameters. An IDL interface provides the information needed to develop clients that use the interface's operations. Clients are not written in IDL, which is purely a descriptive language, but in languages for which mappings from IDL concepts have been defined. The mapping of an IDL concept to a client language construct will depend on the facilities available in the client language. For example, an IDL exception might be mapped to a structure in a language that has no notion of exception, or to an exception in a language that does. The binding of IDL concepts to the C language is described in Chapter 5; the C language has no special status except that it is the first language for which IDL mappings have been established.

IDL obeys the same lexical rules as C++[1], although new keywords are introduced to support distribution concepts. It also provides full support for standard C++ preprocessing features. The IDL specification is expected to track relevant changes to C++ introduced by the ANSI standardization effort.

The description of IDL's lexical conventions is presented in §4.1 on page 47. A description of IDL preprocessing is presented in §4.2 on page 54. The scope rules for identifiers in an IDL specification are described in §4.11 on page 76.

The IDL grammar is a subset of ANSI C++ with additional constructs to support the operation invocation mechanism. IDL is a declarative language; it supports C++ syntax for constant, type, and operation declarations; it does not include any algorithmic structures or variables. The grammar is presented in §4.3 on page 54.

IDL-specific pragmas (those not defined for C++) may appear anywhere in a specification; the textual location of these pragmas may be semantically constrained by a particular implementation.

A source file containing interface specifications written in IDL must have a ".idl" extension. The file orb.idl, which contains IDL type definitions and is available on every ORB implementation, is described in Appendix A.

This chapter describes IDL semantics and gives the syntax for IDL grammatical constructs. The description of IDL grammar uses a syntax notation that is similar to Extended Backus-Naur format (EBNF); TBL. 1 on page 46 lists the symbols used in this format and their meaning.

TBL. 1 IDL EBNF Format

Symbol	Meaning
::=	Is defined to be
|	Alternatively
<text>	Non-terminal
"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 syntactic unit is optional—may occur zero or one time

1. Ellis, Margaret A. and Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990, ISBN 0-201-51459-1

4.1 Lexical Conventions

This section[2] presents the lexical conventions of IDL. It defines tokens in an IDL specification and describes comments, identifiers, keywords, and literals—integer, character, and floating point constants and string literals.

An IDL specification logically consists of one or more files. A file is conceptually translated in several phases.

The first phase is preprocessing, which performs file inclusion and macro substitution. Preprocessing is controlled by directives introduced by lines having # as the first character other than white space. The result of preprocessing is a sequence of tokens. Such a sequence of tokens, that is, a file after preprocessing, is called a translation unit.

IDL 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. TBL. 2 on page 47 shows the IDL alphabetic characters; upper- and lower-case equivalencies are paired.

TBL. 2 The 114 Alphabetic Characters (Letters)

Char.	Description	Char.	Description
Aa	Upper/Lower-case A	Àà	Upper/Lower-case A with grave accent
Bb	Upper/Lower-case B	Áá	Upper/Lower-case A with acute accent
Cc	Upper/Lower-case C	Ââ	Upper/Lower-case A with circumflex accent
Dd	Upper/Lower-case D	Ãã	Upper/Lower-case A with tilde
Ee	Upper/Lower-case E	Ää	Upper/Lower-case A with diaeresis
Ff	Upper/Lower-case F	Åå	Upper/Lower-case A with ring above
Gg	Upper/Lower-case G	Ææ	Upper/Lower-case dipthong A with E
Hh	Upper/Lower-case H	Çç	Upper/Lower-case C with cedilla
Ii	Upper/Lower-case I	Èè	Upper/Lower-case E with grave accent
Jj	Upper/Lower-case J	Éé	Upper/Lower-case E with acute accent
Kk	Upper/Lower-case K	Êê	Upper/Lower-case E with circumflex accent
Ll	Upper/Lower-case L	Ëë	Upper/Lower-case E with diaeresis
Mm	Upper/Lower-case M	Ìì	Upper/Lower-case I with grave accent
Nn	Upper/Lower-case N	Íí	Upper/Lower-case I with acute accent
Oo	Upper/Lower-case O	Îî	Upper/Lower-case I with circumflex accent

2. This section is an adaptation of *The Annotated C++ Reference Manual,* Chapter 2; it differs in the list of legal keywords and punctuation.

TBL. 2 The 114 Alphabetic Characters (Letters) (Continued)

Char.	Description	Char.	Description
Pp	Upper/Lower-case P	Ïï	Upper/Lower-case I with diaeresis
Qq	Upper/Lower-case Q	††	Upper/Lower-case Icelandic eth
Rr	Upper/Lower-case R	Ññ	Upper/Lower-case N with tilde
Ss	Upper/Lower-case S	Òò	Upper/Lower-case O with grave accent
Tt	Upper/Lower-case T	Óó	Upper/Lower-case O with acute accent
Uu	Upper/Lower-case U	Ôô	Upper/Lower-case O with circumflex accent
Vv	Upper/Lower-case V	Õõ	Upper/Lower-case O with tilde
Ww	Upper/Lower-case W	Öö	Upper/Lower-case O with diaeresis
Xx	Upper/Lower-case X	Øø	Upper/Lower-case O with oblique stroke
Yy	Upper/Lower-case Y	Ùù	Upper/Lower-case U with grave accent
Zz	Upper/Lower-case Z	Úú	Upper/Lower-case U with acute accent
		Ûû	Upper/Lower-case U with circumflex accent
		Üü	Upper/Lower-case U with diaeresis
		††	Upper/Lower-case Y with acute accent
		††	Upper/Lower-case Icelandic thorn
		ß	Lower-case German sharp S
		ÿ	Lower-case Y with diaeresis

† *denotes character unprintable in this document*

TBL. 3 on page 48 lists the decimal digit characters.

TBL. 3 Decimal Digits

0 1 2 3 4 5 6 7 8 9

TBL. 4 on page 48 shows the graphic characters.

TBL. 4 The 65 Graphic Characters

Char.	Description	Char.	Description
!	exclamation point	¡	inverted exclamation mark
"	double quote	¢	cent sign
#	number sign	£	pound sign
$	dollar sign	¤	currency sign
%	percent sign	¥	yen sign
&	ampersand	†	broken bar

TBL. 4 The 65 Graphic Characters (Continued)

Char.	Description	Char.	Description
'	apostrophe	§	section/paragraph sign
(	left parenthesis	¨	diaeresis
)	right parenthesis	©	copyright sign
*	asterisk	ª	feminine ordinal indicator
+	plus sign	«	left angle quotation mark
,	comma	¬	not sign
-	hyphen, minus sign	†	soft hyphen
.	period, full stop	®	registered trade mark sign
/	solidus	¯	macron
:	colon	°	ring above, degree sign
;	semicolon	±	plus-minus sign
<	less-than sign	²	superscript two
=	equals sign	³	superscript three
>	greater-than sign	´	acute
?	question mark	µ	micro
@	commercial at	¶	pilcrow
[	left square bracket	•	middle dot
\	reverse solidus	¸	cedilla
]	right square bracket	¹	superscript one
^	circumflex	º	masculine ordinal indicator
_	low line, underscore	»	right angle quotation mark
‘	grave	†	vulgar fraction 1/4
{	left curly bracket	†	vulgar fraction 1/2
|	vertical line	†	vulgar fraction 3/4
}	right curly bracket	¿	inverted question mark
~	tilde	×	multiplication sign
		÷	division sign

† *denotes character unprintable in this document.*

The formatting characters are shown in TBL. 5 on page 50.

TBL. 5 The Formatting Characters

Description	Abbreviation	ISO 646 Octal Value
alert	BEL	007
backspace	BS	010
horizontal tab	HT	011
newline	NL, LF	012
vertical tab	VT	013
form feed	FF	014
carriage return	CR	015

4.1.1 Tokens

There are five kinds of tokens: identifiers, keywords, literals, operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments (collective, "white space"), 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.

4.1.2 Comments

The characters /* start a comment, which terminates with the characters */. These comments do not nest. The characters // start a comment, which terminates at the end of the line on which they occur. The comment characters //, /*, and */ have no special meaning within a // comment and are treated just like other characters. Similarly, the comment characters // and /* have no special meaning within a /* comment. Comments may contain alphabetic, digit, graphic, space, horizontal tab, vertical tab, form feed and newline characters.

4.1.3 Identifiers

An identifier is an arbitrarily long sequence of alphabetic, digit, and underscore ("_") characters. The first character must be an alphabetic character. All characters are significant.

Identifiers that differ only in case collide and yield a compilation error. An identifier for a definition must be spelled consistently (with respect to case) throughout a specification.

When comparing two identifiers to see if they collide:

- Upper- and lower-case letters are treated as the same letter. TBL. 2 on page 47 defines the equivalence mapping of upper- and lower-case letters.
- The comparison does *not* take into account equivalences between digraphs and pairs of letters (e.g., “æ” and “ae” are not considered equivalent) or equivalences between accented and non-accented letters (e.g., “Á” and “A” are not considered equivalent).
- All characters are significant.

There is only one namespace for IDL identifiers. Using the same identifier for a constant and an interface, for example, produces a compilation error.

4.1.4 Keywords

The identifiers listed in TBL. 6 on page 51 are reserved for use as keywords, and may not be used otherwise.

TBL. 6 Keywords

any	default	interface	readonly	unsigned
attribute	double	long	sequence	union
boolean	enum	module	short	void
case	exception	octet	string	FALSE
char	float	oneway	struct	Object
const	in	out	switch	TRUE
context	inout	raises	typedef	

Keywords obey the rules for identifiers (see §4.1.3 on page 50) and must be written exactly as shown in the above list. For example, “**boolean**” is correct; “**Boolean**” produces a compilation error.

IDL specifications use the characters shown in TBL. 7 on page 51 as punctuation.

TBL. 7 Punctuation Characters

<table>
<tr><td>;</td><td>{</td><td>}</td><td>:</td><td>,</td><td>=</td><td>+</td><td>-</td><td>(</td><td>)</td><td><</td><td>></td><td>[</td><td>]</td></tr>
<tr><td>'</td><td>"</td><td>\</td><td>|</td><td>^</td><td>&</td><td>*</td><td>/</td><td>%</td><td>~</td><td></td><td></td><td></td><td></td></tr>
</table>

In addition, the tokens listed in TBL. 8 on page 52 are used by the preprocessor.

TBL. 8 Preprocessor Tokens

#	##	!	||	&&

4.1.5 Literals

4.1.5.1 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.

4.1.5.2 Character Literals

A character literal is one or more characters enclosed in single quotes, as in ’x’. Character literals have type **char**.

A character is an 8-bit quantity with a numerical value between 0 and 255 (decimal). The value of a space, alphabetic, digit or graphic character literal is the numerical value of the character as defined in the ISO Latin-1 (8859.1) character set standard (See TBL. 2 on page 47, TBL. 3 on page 48, and TBL. 4 on page 48). The value of a 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 TBL. 5 on page 50). The meaning of all other characters is implementation-dependent.

Nongraphic characters must be represented using escape sequences as defined below in TBL. 9 on page 52. Note that escape sequences must be used to represent single quote and backslash characters in character literals.

TBL. 9 Escape Sequences

Description	Escape Sequence
newline	\n
horizontal tab	\t
vertical tab	\v
backspace	\b
carriage return	\r

TBL. 9 Escape Sequences (Continued)

Description	Escape Sequence
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 one 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.

4.1.5.3 Floating-point Literals

A floating-point literal consists of an integer part, a decimal point, a fraction part, an e or E, an optionally signed integer exponent, and an optional type suffix. 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.

4.1.5.4 String Literals

A string literal is a sequence of characters (as defined in §4.1.5.2 on page 52) surrounded by double quotes, as in "...".

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'.

4.2 Preprocessing

IDL preprocessing, which is based on ANSI C++ preprocessing, provides macro substitution, conditional compilation, and source file inclusion. In addition, directives are provided to control line numbering in diagnostics and for symbolic debugging, to generate a diagnostic message with a given token sequence, and to perform implementation-dependent actions (the **#pragma** directive). Certain predefined names are available. These facilities are conceptually handled by a preprocessor, which may or may not actually be implemented as a separate process.

Lines beginning with # (also called "directives") communicate with this preprocessor. White space may appear before the #. These lines have syntax independent of the rest of IDL; they may appear anywhere and have effects that last (independent of the IDL scoping rules) until the end of the translation unit. The textual location of IDL-specific pragmas may be semantically constrained.

A preprocessing directive (or any line) may be continued on the next line in a source file by placing a backslash character ("\"), immediately before the newline at the end of the line to be continued. The preprocessor effects the continuation by deleting the backslash and the newline before the input sequence is divided into tokens. A backslash character may not be the last character in a source file.

A preprocessing token is an IDL token (§4.1.1 on page 50), a file name as in a **#include** directive, or any single character, other than white space, that does not match another preprocessing token.

The primary use of the preprocessing facilities is to include definitions from other IDL specifications. Text in files included with a **#include** directive is treated as if it appeared in the including file. A complete description of the preprocessing facilities may be found in *The Annotated C++ Reference Manual*, Chapter 16.

4.3 IDL Grammar

(1) <specification> ::= **<definition>**$^{+}$

(2) <definition> ::= **<type_dcl>** ";"
| **<const_dcl>** ";"

| <except_dcl> “;”
| <interface> “;”
| <module> “;”

(3) <module> ::= “module” <identifier> “{“ <definition>+ “}”

(4) <interface> ::= <interface_dcl>
| <forward_dcl>

(5) <interface_dcl> ::= <interface_header> “{” <interface_body> “}”

(6) <forward_dcl> ::= “interface” <identifier>

(7) <interface_header> ::= “interface” <identifier> [<inheritance_spec>]

(8) <interface_body> ::= <export>*

(9) <export> ::= <type_dcl> “;”
| <const_dcl> “;”
| <except_dcl> “;”
| <attr_dcl> “;”
| <op_dcl> “;”

(10) <inheritance_spec> ::= “:” <scoped_name> { “,” <scoped_name> }*

(11) <scoped_name> ::= <identifier>
| “::” <identifier>
| <scoped_name> “::” <identifier>

(12) <const_dcl> ::= “const” <const_type> <identifier> “=” <const_exp>

(13) <const_type> ::= <integer_type>
| <char_type>
| <boolean_type>
| <floating_pt_type>
| <string_type>
| <scoped_name>

(14) <const_exp> ::= <or_expr>

(15) <or_expr> ::= <xor_expr>
| <or_expr> “|” <xor_expr>

(16) <xor_expr> ::= <and_expr>
| <xor_expr> “^” <and_expr>

(17) <and_expr> ::= <shift_expr>
| <and_expr> “&” <shift_expr>

(18) <shift_expr> ::= <add_expr>
| <shift_expr> “>>” <add_expr>
| <shift_expr> “<<” <add_expr>

(19) <add_expr> ::= <mult_expr>
| <add_expr> “+” <mult_expr>
| <add_expr> “-” <mult_expr>

(20) <mult_expr> ::= <unary_expr>
| <mult_expr> “*” <unary_expr>

```
                              |    <mult_expr> "/" <unary_expr>
                              |    <mult_expr> "%" <unary_expr>
(21) <unary_expr>             ::=  <unary_operator> <primary_expr>
                              |    <primary_expr>
(22) <unary_operator>         ::=  "-"
                              |    "+"
                              |    "~"
(23) <primary_expr>           ::=  <scoped_name>
                              |    <literal>
                              |    "(" <const_exp> ")"
(24) <literal>                ::=  <integer_literal>
                              |    <string_literal>
                              |    <character_literal>
                              |    <floating_pt_literal>
                              |    <boolean_literal>
(25) <boolean_literal>        ::=  "TRUE"
                              |    "FALSE"
(26) <positive_int_const>     ::=  <const_exp>
(27) <type_dcl>               ::=  "typedef" <type_declarator>
                              |    <struct_type>
                              |    <union_type>
                              |    <enum_type>
(28) <type_declarator>        ::=  <type_spec> <declarators>
(29) <type_spec>              ::=  <simple_type_spec>
                              |    <constr_type_spec>
(30) <simple_type_spec>       ::=  <base_type_spec>
                              |    <template_type_spec>
                              |    <scoped_name>
(31) <base_type_spec>         ::=  <floating_pt_type>
                              |    <integer_type>
                              |    <char_type>
                              |    <boolean_type>
                              |    <octet_type>
                              |    <any_type>
(32) <template_type_spec>     ::=  <sequence_type>
                              |    <string_type>
(33) <constr_type_spec>       ::=  <struct_type>
                              |    <union_type>
                              |    <enum_type>
(34) <declarators>            ::=  <declarator> { "," <declarator> }*
(35) <declarator>             ::=  <simple_declarator>
                              |    <complex_declarator>
```

```
(36) <simple_declarator>   ::=  <identifier>

(37) <complex_declarator>  ::=  <array_declarator>

(38) <floating_pt_type>    ::=  "float"
                           |    "double"

(39) <integer_type>        ::=  <signed_int>
                           |    <unsigned_int>

(40) <signed_int>          ::=  <signed_long_int>
                           |    <signed_short_int>

(41) <signed_long_int>     ::=  "long"

(42) <signed_short_int>    ::=  "short"

(43) <unsigned_int>        ::=  <unsigned_long_int>
                           |    <unsigned_short_int>

(44) <unsigned_long_int>   ::=  "unsigned" "long"

(45) <unsigned_short_int>  ::=  "unsigned" "short"

(46) <char_type>           ::=  "char"

(47) <boolean_type>        ::=  "boolean"

(48) <octet_type>          ::=  "octet"

(49) <any_type>            ::=  "any"

(50) <struct_type>         ::=  "struct" <identifier> "{" <member_list> "}"

(51) <member_list>         ::=  <member>+

(52) <member>              ::=  <type_spec> <declarators> ";"

(53) <union_type>          ::=  "union" <identifier> "switch" "(" <switch_type_spec> ")"
                                    "{" <switch_body> "}"

(54) <switch_type_spec>    ::=  <integer_type>
                           |    <char_type>
                           |    <boolean_type>
                           |    <enum_type>
                           |    <scoped_name>

(55) <switch_body>         ::=  <case>+

(56) <case>                ::=  <case_label>+ <element_spec> ";"

(57) <case_label>          ::=  "case" <const_exp> ":"
                           |    "default" ":"

(58) <element_spec>        ::=  <type_spec> <declarator>

(59) <enum_type>           ::=  "enum" <identifier> "{" <enumerator> { "," <enumerator> }* "}"

(60) <enumerator>          ::=  <identifier>

(61) <sequence_type>       ::=  "sequence" "<" <simple_type_spec> "," <positive_int_const> ">"
                           |    "sequence" "<" <simple_type_spec> ">"
```

(62) <string_type> ::= "string" "<" <positive_int_const> ">"
| "string"

(63) <array_declarator> ::= <identifier> <fixed_array_size>$^{+}$

(64) <fixed_array_size> ::= "[" <positive_int_const> "]"

(65) <attr_dcl> ::= ["readonly"] "attribute" <simple_type_spec> <declarators>

(66) <except_dcl> ::= "exception" <identifier> "{" <member>* "}"

(67) <op_dcl> ::= [<op_attribute>] <op_type_spec> <identifier> <parameter_dcls>
[<raises_expr>] [<context_expr>]

(68) <op_attribute> ::= "oneway"

(69) <op_type_spec> ::= <simple_type_spec>
| "void"

(70) <parameter_dcls> ::= "(" <param_dcl> { "," <param_dcl> }* ")"
| "(" ")"

(71) <param_dcl> ::= <param_attribute> <simple_type_spec> <declarator>

(72) <param_attribute> ::= "in"
| "out"
| "inout"

(73) <raises_expr> ::= "raises" "(" <scoped_name> { "," <scoped_name> }* ")"

(74) <context_expr> ::= "context" "(" <string_literal> { "," <string_literal> }* ")"

4.4 IDL Specification

An IDL specification consists of one or more type definitions, constant definitions, exception definitions, or module definitions. The syntax is:

<specification> ::= <definition>$^{+}$

<definition> ::= <type_dcl> ";"
| <const_dcl> ";"
| <except_dcl> ";"
| <interface> ";"
| <module> ";"

See §4.6 on page 62, §4.7 on page 65, and §4.8 on page 72, respectively, for specifications of **<const_dcl>**, **<type_dcl>**, and **<except_dcl>**.

4.4.1 Module Declaration

A module definition satisfies the following syntax:

<module> ::= "module" <identifier> "{" <definition>$^{+}$ "}"

The module construct is used to scope IDL identifiers; see §4.11 on page 76 for details.

4.4.2 Interface Declaration

An interface definition satisfies the following syntax:

```
<interface>            ::=  <interface_dcl>
                       |    <forward_dcl>

<interface_dcl>        ::=  <interface_header> "{" <interface_body> "}"

<forward_dcl>          ::=  "interface" <identifier>

<interface_header>     ::=  "interface" <identifier> [ <inheritance_spec> ]

<interface_body>       ::=  <export>*

<export>               ::=  <type_dcl> ";"
                       |    <const_dcl> ";"
                       |    <except_dcl> ";"
                       |    <attr_dcl> ";"
                       |    <op_dcl> ";"
```

4.4.2.1 Interface Header

The interface header consists of two elements:

- The interface name. The name must be preceded by the keyword **interface**, and consists of an identifier that names the interface.
- An optional inheritance specification. The inheritance specification is described in §4.4.2.2 on page 59.

The **<identifier>** that names an interface defines a legal type name. Such a type name may be used anywhere an **<identifier>** is legal in the grammar, subject to semantic constraints as described in the following sections. Since one can only hold references to an object, the meaning of a parameter or structure member which is an interface type is as a *reference* to an object supporting that interface. Each language binding describes how the programmer must represent such interface references.

4.4.2.2 Inheritance Specification

The syntax for inheritance is as follows:

```
<inheritance_spec>     ::= ":" <scoped_name> {"," <scoped_name>}*

<scoped_name>          ::= <identifier>
                       | "::" <identifier>
                       | <scoped_name> "::" <identifier>
```

Each **<scoped_name>** in an **<inheritance_spec>** must denote a previously defined interface. See §4.5 on page 60 for the description of inheritance.

4.4.2.3 Interface Body

The interface body contains the following kinds of declarations:

- Constant declarations, which specify the constants that the interface exports; constant declaration syntax is described in §4.6 on page 62.
- Type declarations, which specify the type definitions that the interface exports; type declaration syntax is described in §4.7 on page 65.
- Exception declarations, which specify the exception structures that the interface exports; exception declaration syntax is described in §4.8 on page 72.
- Attribute declarations, which specify the associated attributes exported by the interface; attribute declaration syntax is described in §4.10 on page 75.
- Operation declarations, which specify the operations that the interface exports and the format of each, including operation name, the type of data returned, the types of all parameters of an operation, legal exceptions which may be returned as a result of an invocation, and contextual information which may affect method dispatch; operation declaration syntax is described in §4.9 on page 72.

Empty interfaces (i.e. those that contain no declarations) are permitted.

Some implementations may require interface-specific pragmas to precede the interface body.

4.4.2.4 Forward Declaration

A forward declaration declares the name of an interface without defining it. This permits the definition of interfaces that refer to each other. The syntax consists simply of the keyword **interface** followed by an **<identifier>** that names the interface. The actual definition must follow later in the specification.

Multiple forward declarations of the same interface name are legal.

4.5 Inheritance

An interface can be derived from another interface, which is then called a *base* interface of the derived interface. A derived interface, like all interfaces, may declare new elements (constants, types, attributes, exceptions, and operations). In addition, unless redefined in the derived interface, the elements of a base interface can be referred to as if they were elements of the derived interface. The name resolution operator ("::") may be used to refer to a base element explicitly; this permits reference to a name that has been redefined in the derived interface.

A derived interface may redefine any of the type, constant, and exception names which have been inherited; the scope rules for such names are described in §4.11 on page 76.

An interface is called a direct base if it is mentioned in the **<inheritance_spec>** and an indirect base if it is not a direct base but is a base interface of one of the interfaces mentioned in the **<inheritance_spec>**.

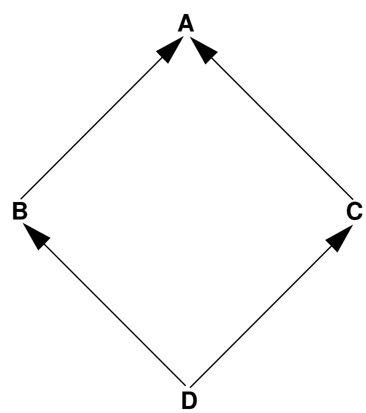
An interface may be derived from any number of base interfaces. Such use of more than one direct base interface is often called multiple inheritance. The order of derivation is not significant.

An interface may not be specified as a direct base interface of a derived interface more than once; it may be an indirect base interface more than once. Consider the following example:

```
interface A { ... }
interface B: A { ... }
interface C: A { ... }
interface D: B, C { ... }
```

The relationships between these interfaces is shown in FIG. 11 on page 61. This "diamond" shape is legal.

FIG. 11 Legal Multiple Inheritance Example



Reference to base interface elements must be unambiguous. Reference to a base interface element is ambiguous if the expression used refers to a constant, type, or exception in more than one base interface. (It is currently illegal to inherit from two interfaces with the same operation or attribute name, or to redefine an operation or attribute name in the derived interface.) Ambiguities can be resolved by qualifying a name with its interface name (i.e., using a **<scoped_name>**).

References to constants, types, and exceptions are bound to an interface when it is defined i.e., replaced with the equivalent global **<scoped_name>**s. This guarantees that the syntax and semantics of an interface are not changed when the interface is a base class for a derived class. Consider the following example:

```
const long L = 3;

interface A {
    void f (in float s[L]);              // s has 3 floats
};

interface B {
    const long L = 4;
};

interface C: B, A {}                     // what is f()'s signature?
```

The early binding of constants, types, and exceptions at interface definition guarantees that the signature of operation **f** in interface **C** is

```
void f(in float s[3]);
```

which is identical to that in interface **A**. This rule also prevents redefinition of a constant, type, or exception in the derived interface from affecting the operations and attributes inherited from a base interface.

Interface inheritance causes all identifiers in the closure of the inheritance tree to be imported into the current naming scope. A type name, constant name, enumeration value name, or exception name from an enclosing scope can be redefined in the current scope. An attempt to use an ambiguous name without qualification is a compilation error.

Operation names are used at runtime by both the stub and dynamic interfaces. As a result, all operations that might apply to a particular object must have unique names. This requirement prohibits redefining an operation name in a derived class, as well as inheriting two operations with the same name.

NOTE *It is anticipated that future revisions of the language may relax this rule in some way, perhaps allowing overloading or providing some means to distinguish operations with the same name.*

4.6 Constant Declaration

This section describes the syntax for constant declarations.

4.6.1 Syntax

The syntax for a constant declaration is:

```
<const_dcl>          ::=  “const” <const_type> <identifier> “=” <const_exp>

<const_type>         ::=  <integer_type>
                     |    <char_type>
                     |    <boolean_type>
                     |    <floating_pt_type>
                     |    <string_type>
                     |    <scoped_name>

<const_exp>          ::=  <or_expr>

<or_expr>            ::=  <xor_expr>
                     |    <or_expr> “|” <xor_expr>

<xor_expr>           ::=  <and_expr>
                     |    <xor_expr> “^” <and_expr>

<and_expr>           ::=  <shift_expr>
                     |    <and_expr> “&” <shift_expr>

<shift_expr>         ::=  <add_expr>
                     |    <shift_expr> “>>” <add_expr>
                     |    <shift_expr> “<<” <add_expr>

<add_expr>           ::=  <mult_expr>
                     |    <add_expr> “+” <mult_expr>
                     |    <add_expr> “-” <mult_expr>

<mult_expr>          ::=  <unary_expr>
                     |    <mult_expr> “*” <unary_expr>
                     |    <mult_expr> “/” <unary_expr>
                     |    <mult_expr> “%” <unary_expr>

<unary_expr>         ::=  <unary_operator> <primary_expr>
                     |    <primary_expr>

<unary_operator>     ::=  “-”
                     |    “+”
                     |    “~”

<primary_expr>       ::=  <scoped_name>
                     |    <literal>
                     |    “(” <const_exp> “)”

<literal>            ::=  <integer_literal>
                     |    <string_literal>
                     |    <character_literal>
                     |    <floating_pt_literal>
                     |    <boolean_literal>

<boolean_literal>    ::=  “TRUE”
                     |    “FALSE”
```

<positive_int_const> ::= **<const_exp>**

4.6.2 Semantics

The **<scoped_name>** in the **<const_type>** production must be a previously defined name of an **<integer_type>**, **<char_type>**, **<boolean_type>**, **<floating_pt_type>**, or **<string_type>** constant.

Mixed type expressions (e.g. integers mixed with floats) are illegal.

An integer constant expression is evaluated as unsigned long unless it contains a negated integer literal or the name of an integer constant with a negative value. In the latter case, the constant expression is evaluated as signed long. The computed value is coerced back to the target type in constant initializers. It is an error if the computed value exceeds the precision of the target type. It is an error if any intermediate value exceeds the range of the evaluated-as type (long or unsigned long).

All floating-point literals are double, all floating-point constants are coerced to double, and all floating-point expressions are computed as doubles. The computed double value is coerced back to the target type in constant initializers. It is an error if this coercion fails or if any intermediate values (when evaluating the expression) exceed the range of double.

Unary (+ –) and binary (* / + –) operators are applicable in floating-point expressions. Unary (+ – ~) and binary (* / % + – << >> & | ^) operators are applicable in integer expressions.

The “~” unary operator indicates that the bit-complement of the expression to which it is applied should be generated. For the purposes of such expressions, the values are 2’s complement numbers. As such, the complement can be generated as follows:

long	–(value+1)
unsigned long	(2**32 – 1) – value

The “%” binary operator yields the remainder from the division of the first expression by the second. If the second operand of “%” is 0, the result is undefined; otherwise

(a/b)*b + a%b

is equal to a. If both operands are nonnegative, then the remainder is nonnegative; if not, the sign of the remainder is implementation dependent.

The “<<”binary operator indicates that the value of the left operand should be shifted left the number of bits specified by the right operand, with 0 fill for the vacated bits. The right operand must be in the range 0 <= right operand < 32.

The “>>” binary operator indicates that the value of the left operand should be shifted right the number of bits specified by the right operand, with 0 fill for the vacated bits. The right operand must be in the range 0 <= right operand < 32.

The “&” binary operator indicates that the logical, bitwise AND of the left and right operands should be generated.

The “|” binary operator indicates that the logical, bitwise OR of the left and right operands should be generated.

The “^” binary operator indicates that the logical, bitwise EXCLUSIVE-OR of the left and right operands should be generated.

<positive_int_const> must evaluate to a positive integer constant.

4.7 Type Declaration

IDL provides constructs for naming data types; that is, it provides C language-like declarations that associate an identifier with a type. IDL uses the **typedef** keyword to associate a name with a data type; a name is also associated with a data type via the **struct**, **union**, and **enum** declarations; the syntax is:

```
<type_dcl>            ::=  “typedef” <type_declarator>
                       |   <struct_type>
                       |   <union_type>
                       |   <enum_type>

<type_declarator>     ::=  <type_spec> <declarators>
```

For type declarations, IDL defines a set of type specifiers to represent typed values. The syntax is as follows:

```
<type_spec>           ::=  <simple_type_spec>
                       |   <constr_type_spec>

<simple_type_spec>    ::=  <base_type_spec>
                       |   <template_type_spec>
                       |   <scoped_name>

<base_type_spec>      ::=  <floating_pt_type>
                       |   <integer_type>
                       |   <char_type>
                       |   <boolean_type>
                       |   <octet_type>
                       |   <any_type>

<template_type_spec>  ::=  <sequence_type>
                       |   <string_type>
```

```
<constr_type_spec>      ::=  <struct_type>
                        |    <union_type>
                        |    <enum_type>
<declarators>           ::=  <declarator> { "," <declarator> }*
<declarator>            ::=  <simple_declarator>
                        |    <complex_declarator>
<simple_declarator>     ::=  <identifier>
<complex_declarator>    ::=  <array_declarator>
```

The **<scoped_name>** in **<simple_type_spec>** must be a previously defined type.

As seen above, IDL type specifiers consist of scalar data types and type constructors. IDL type specifiers can be used in operation declarations to assign data types to operation parameters. The next sections describe basic and constructed type specifiers.

4.7.1 Basic Types

The syntax for the supported basic types is as follows:

```
<floating_pt_type>      ::=  "float"
                        |    "double"
<integer_type>          ::=  <signed_int>
                        |    <unsigned_int>
<signed_int>            ::=  <signed_long_int>
                        |    <signed_short_int>
<signed_long_int>       ::=  "long"
<signed_short_int>      ::=  "short"
<unsigned_int>          ::=  <unsigned_long_int>
                        |    <unsigned_short_int>
<unsigned_long_int>     ::=  "unsigned" "long"
<unsigned_short_int>    ::=  "unsigned" "short"
<char_type>             ::=  "char"
<boolean_type>          ::=  "boolean"
<octet_type>            ::=  "octet"
<any_type>              ::=  "any"
```

Each IDL data type is mapped to a native data type via the appropriate language mapping. Conversion errors between IDL data types and the native types to which they are mapped can occur during the performance of an operation invocation. The invocation mechanism (client stub, dynamic invocation engine, and skeletons) may signal an exception condition to the client if an attempt is made to convert an illegal value. The standard exceptions which are to be signalled in such situations are defined in §4.13 on page 79.

4.7.1.1 Integer Types

IDL supports **long** and **short** signed and **unsigned** integer data types. **long** represents the range $-2^{31} .. 2^{31} - 1$ while **unsigned long** represents the range $0 .. 2^{32} - 1$. **short** represents the range $-2^{15} .. 2^{15} - 1$, while **unsigned short** represents the range $0 .. 2^{16} - 1$.

4.7.1.2 Floating-Point Types

IDL floating-point types are **float** and **double**. The **float** type represents IEEE single-precision floating point numbers; the **double** type represents IEEE double-precision floating point numbers.The IEEE floating point standard specification (*IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985) should be consulted for more information on the precision afforded by these types.

Implementations that do not fully support the value set of the IEEE 754 floating-point standard must completely specify their deviance from the standard.

4.7.1.3 Char Type

IDL defines a **char** data type consisting of 8-bit quantities.

The ISO Latin-1 (8859.1) character set standard defines the meaning and representation of all possible graphic characters (i.e., the space, alphabetic, digit and graphic characters defined in TBL. 2 on page 47, TBL. 3 on page 48, and TBL. 4 on page 48). The meaning and representation of the null and formatting characters (see TBL. 5 on page 50) is the numerical value of the character as defined in the ASCII (ISO 646) standard. The meaning of all other characters is implementation-dependent.

During transmission, characters may be converted to other appropriate forms as required by a particular language binding. Such conversions may change the representation of a character but maintain the character's meaning. For example, a character may be converted to and from the appropriate representation in international character sets.

4.7.1.4 Boolean Type

The **boolean** data type is used to denote a data item that can only take one of the values TRUE and FALSE.

4.7.1.5 Octet Type

The **octet** type is an 8-bit quantity that is guaranteed not to undergo any conversion when transmitted by the communication system.

4.7.1.6 Any Type

The **any** type permits the specification of values that can express any IDL type. If used in a request, an **any** must satisfy the constraints specified in Appendix A.

4.7.2 Constructed Types

The constructed types are:

```
<constr_type_spec>      ::=  <struct_type>
                        |    <union_type>
                        |    <enum_type>
```

Although it is syntactically possible to generate recursive type specifications in IDL, such recursion is semantically constrained. The only permissible form of recursive type specification is through the use of the **sequence** template type. For example, the following is legal:

```
struct foo {
    long value;
    sequence<foo> chain;
}
```

See §4.7.3.1 on page 70 for details of the **sequence** template type.

4.7.2.1 Structures

The structure syntax is:

```
<struct_type>           ::=  "struct" <identifier> "{" <member_list> "}"

<member_list>           ::=  <member>+

<member>                ::=  <type_spec> <declarators> ";"
```

The **<identifier>** in **<struct_type>** defines a new legal type. Structure types may also be named using a **typedef** declaration.

Name scoping rules require that the member declarators in a particular structure be unique. The value of a **struct** is the value of all of its members.

4.7.2.2 Discriminated Unions

The discriminated **union** syntax is:

```
<union_type>            ::=  "union" <identifier> "switch" "(" <switch_type_spec> ")"
                                 "{" <switch_body> "}"

<switch_type_spec>      ::=  <integer_type>
                        |    <char_type>
                        |    <boolean_type>
```

	|	**<enum_type>**
	|	**<scoped_name>**
<switch_body>	::=	**<case>**$^+$
<case>	::=	**<case_label>**$^+$ **<element_spec>** “;”
<case_label>	::=	“case” **<const_exp>** “:”
	|	“default” “:”
<element_spec>	::=	**<type_spec> <declarator>**

IDL unions are a cross between the C **union** and **switch** statements. IDL unions must be discriminated; that is, the union header must specify a typed tag field that determines which union member to use for the current instance of a call. The **<identifier>** following the **union** keyword defines a new legal type. Union types may also be named using a **typedef** declaration. The **<const_exp>** in a **<case_label>** must be consistent with the **<switch_type_spec>**. A **default** case can appear at most once. The **<scoped_name>** in the **<switch_type_spec>** production must be a previously defined **integer**, **char**, **boolean** or **enum** type.

Case labels must match or be automatically castable to the defined type of the discriminator. The complete set of matching rules are shown in TBL. 10 on page 69.

TBL. 10 Case Label Matching

Discriminator Type	Matched By
long	any integer value in the value range of long
short	any integer value in the value range of short
unsigned long	any integer value in the value range of unsigned long
unsigned short	any integer value in the value range of unsigned short
char	char
boolean	TRUE or FALSE
enum	any enumerator for the discriminator enum type

Name scoping rules require that the element declarators in a particular union be unique. If the **<switch_type_spec>** is an **<enum_type>**, the identifier for the enumeration is in the scope of the union; as a result, it must be distinct from the element declarators.

It is not required that all possible values of the union discriminator be listed in the **<switch_body>**. The value of a union is the value of the discriminator together with one of the following:

- if the discriminator value was explicitly listed in a **case** statement, the value of the element associated with that **case** statement;

- if a default **case** label was specified, the value of the element associated with the default **case** label;
- no additional value.

Access to the discriminator and the related element is language-mapping dependent.

4.7.2.3 Enumerations

Enumerated types consist of ordered lists of identifiers. The syntax is:

<enum_type> ::= **"enum" <identifier> "{" <enumerator> { "," <enumerator> }* "}"**

<enumerator> ::= **<identifier>**

A maximum of 2^{32} identifiers may be specified in an enumeration; as such, the enumerated names must be mapped to a native data type capable of representing a maximally-sized enumeration. The order in which the identifiers are named in the specification of an enumeration defines the relative order of the identifiers. Any language mapping which permits two enumerators to be compared or defines successor/predecessor functions on enumerators must conform to this ordering relation. The **<identifier>** following the **enum** keyword defines a new legal type. Enumerated types may also be named using a **typedef** declaration.

4.7.3 Template Types

The template types are:

<template_type_spec> ::= **<sequence_type>**
| **<string_type>**

4.7.3.1 Sequences

IDL defines the sequence type **sequence**. A sequence is a one-dimensional array with two characteristics: a maximum size (which is fixed at compile time) and a length (which is determined at run time).

The syntax is:

<sequence_type> ::= **"sequence" "<" <simple_type_spec> "," <positive_int_const> ">"**
| **"sequence" "<" <simple_type_spec> ">"**

The second parameter in a sequence declaration indicates the maximum size of the sequence. If a positive integer constant is specified for the maximum size, the sequence is termed a bounded sequence. Prior to passing a bounded sequence as a function argument (or as a field in a structure or union), the length of the sequence must be set in a language-mapping dependent manner. After receiving a sequence result from an operation invocation, the length of the returned sequence will have been set; this value may be obtained in a language-mapping dependent manner.

If no maximum size is specified, size of the sequence is unspecified (unbounded). Prior to passing such a sequence as a function argument (or as a field in a structure or union), the length of the sequence, the maximum size of the sequence, and the address of a buffer to hold the sequence must be set in a language-mapping dependent manner. After receiving such a sequence result from an operation invocation, the length of the returned sequence will have been set; this value may be obtained in a language-mapping dependent manner.

4.7.3.2 Strings

IDL defines the string type **string** consisting of all possible 8-bit quantities except null. A string is similar to a sequence of char. As with sequences of any type, prior to passing a string as a function argument (or as a field in a structure or union), the length of the string must be set in a language-mapping dependent manner. The syntax is:

```
<string_type>        ::=  "string" "<" <positive_int_const> ">"
                     |    "string"
```

The argument to the string declaration is the maximum size of the string. If a positive integer maximum size is specified, the string is termed a bounded string; if no maximum size is specified, the string is termed an unbounded string.

Strings are singled out as a separate type because many languages have special built-in functions or standard library functions for string manipulation. A separate string type may permit substantial optimization in the handling of strings compared to what can be done with sequences of general types.

4.7.4 Complex Declarator

4.7.4.1 Arrays

IDL defines multidimensional, fixed-size arrays. An array includes explicit sizes for each dimension.

The syntax for arrays is:

```
<array_declarator>   ::=  <identifier> <fixed_array_size>+

<fixed_array_size>   ::=  "[" <positive_int_const> "]"
```

The array size (in each dimension) is fixed at compile time. When an array is passed as a parameter in an operation invocation, all elements of the array are transmitted.

The implementation of array indices is language mapping specific; passing an array index as a parameter may yield incorrect results.

4.8 Exception Declaration

Exception declarations permit the declaration of struct-like data structures which may be returned to indicate that an exceptional condition has occurred during the performance of a request. The syntax is as follows:

<except_dcl> ::= "exception" <identifier> "{" <member>* "}"

Each exception is characterized by its IDL identifier, an exception type identifier, and the type of the associated return value (as specified by the **<member>**s in its declaration. If an exception is returned as the outcome to a request, then the value of the exception identifier is accessible to the programmer for determining which particular exception was raised.

If an exception is declared with members, a programmer will be able to access the values of those members when an exception is raised. If no members are specified, no additional information is accessible when an exception is raised.

A set of standard exceptions is defined corresponding to standard runtime errors which may occur during the execution of a request. These standard exceptions are documented in §4.13 on page 79.

4.9 Operation Declaration

Operation declarations in IDL are similar to C function declarations. The syntax is:

```
<op_dcl>          ::=  [ <op_attribute> ] <op_type_spec> <identifier> <parameter_dcls>
                            [ <raises_expr> ] [ <context_expr> ]
<op_type_spec>    ::=  <simple_type_spec>
                  |    "void"
```

An operation declaration consists of:

- An optional operation attribute that specifies which invocation semantics the communication system should provide when the operation is invoked. Operation attributes are described in §4.9.1 on page 73.
- The type of the operation's return result; the type may be any type which can be defined in IDL. Operations that do not return a result must specify the **void** type.
- An identifier that names the operation in the scope of the interface in which it is defined.
- A parameter list that specifies zero or more parameter declarations for the operation. Parameter declaration is described in §4.9.2 on page 73.
- An optional raises expression which indicates which exceptions may be raised as a result of an invocation of this operation. Raises expressions are described in Section §4.9.3 on page 74.

- An optional context expression which indicates which elements of the request context may be consulted by the method that implements the operation. Context expressions are described in §4.9.4 on page 74.

Some implementations and/or language mappings may require operation-specific pragmas to immediately precede the affected operation declaration.

4.9.1 Operation Attribute

The operation attribute specifies which invocation semantics the communication service must provide for invocations of a particular operation. An operation attribute is optional. The syntax for its specification is as follows:

```
<op_attribute>          ::=  "oneway"
```

When a client invokes an operation with the **oneway** attribute, the invocation semantics are best-effort, which does not guarantee delivery of the call; best-effort implies that the operation will be invoked at most once. An operation with the **oneway** attribute must not contain any output parameters and must specify a **void** return type. An operation defined with the **oneway** attribute may not include a raises expression; invocation of such an operation, however, may raise a standard exception.

If an **<op_attribute>** is not specified, the invocation semantics is at-most-once if an exception is raised; the semantics are exactly-once if the operation invocation returns successfully.

4.9.2 Parameter Declarations

Parameter declarations in IDL operation declarations have the following syntax:

```
<parameter_dcls>        ::=  "(" <param_dcl> { "," <param_dcl> }* ")"
                        |    "(" ")"

<param_dcl>             ::=  <param_attribute> <simple_type_spec> <declarator>

<param_attribute>       ::=  "in"
                        |    "out"
                        |    "inout"
```

A parameter declaration must have a directional attribute that informs the communication service in both the client and the server of the direction in which the parameter is to be passed. The directional attributes are:

- **in** - the parameter is passed from client to server.
- **out** - the parameter is passed from server to client.
- **inout** - the parameter is passed in both directions.

It is expected that an implementation will *not* attempt to modify an **in** parameter. The ability to even attempt to do so is language-mapping specific; the effect of such an action is undefined.

If an exception is raised as a result of an invocation, the values of the return result and any **out** and **inout** parameters are undefined.

When an unbounded **string** or **sequence** is passed as an **inout** parameter, the returned value cannot be longer than the input value.

4.9.3 Raises Expressions

A **raises** expression specifies which exceptions may be raised as a result of an invocation of the operation. The syntax for its specification is as follows:

<raises_expr> ::= "raises" "(" <scoped_name> { "," <scoped_name> }* ")"

The **<scoped_name>**'s in the **raises** expression must be previously defined exceptions.

In addition to any operation-specific exceptions specified in the **raises** expression, there are a standard set of exceptions that may be signalled by the ORB. These standard exceptions are described in §4.13 on page 79. However, standard exceptions may *not* be listed in a **raises** expression.

The absence of a **raises** expression on an operation implies that there are no operation-specific exceptions. Invocations of such an operation are still liable to receive one of the standard exceptions.

4.9.4 Context Expressions

A **context** expression specifies which elements of the client's context may affect the performance of a request by the object. The syntax for its specification is as follows:

<context_expr> ::= "context" "(" <string_literal> { "," <string_literal> }* ")"

The runtime system guarantees to make the value (if any) associated with each **<string_literal>** in the client's context available to the object implementation when the request is delivered. The ORB and/or object is free to use information in this *request context* during request resolution and performance.

The absence of a context expression indicates that there is no request context associated with requests for this operation.

Each **string_literal** is an arbitrarily long sequence of alphabetic, digit, period ("."), underscore ("_"), and asterisk ("*") characters. The first character of the string must be an alphabetic character. An asterisk may only be used as the last character of the string. Some implementations may use the period character to partition the name space.

The mechanism by which a client associates values with the context identifiers is described in Chapter 6.

4.10 Attribute Declaration

An interface can have attributes as well as operations; as such, attributes are defined as part of an interface. An attribute definition is logically equivalent to declaring a pair of accessor functions; one to retrieve the value of the attribute and one to set the value of the attribute.

The syntax for **attribute** declaration is:

```
<attr_dcl>          ::=   [ "readonly" ] "attribute" <simple_type_spec> <declarators>
```

The optional **readonly** keyword indicates that there is only a single accessor function—the retrieve value function. Consider the following example:

```
interface foo {
    enum material_t {rubber, glass};
    struct position_t {
        float x, y;
    };

    attribute float radius;
    attribute material_t material;
    readonly attribute position_t position;

    •••
};
```

The attribute declarations are equivalent to the following pseudo-specification fragment:

```
•••
float         _get_radius ();
void          _set_radius (in float r);
material_t    _get_material ();
void          _set_material (in material_t m);
position_t    _get_position ();
•••
```

The actual accessor function names are language-mapping specific; the C mappings are described in §5.5 on page 88. Only the attribute name is subject to IDL's name scoping rules; the accessor function names are guaranteed *not* to collide with any legal operation names specifiable in IDL.

Attribute operations return errors by means of standard exceptions.

Attributes are inherited. An attribute name *cannot* be redefined to be a different type. See §4.11 on page 76 for more information on redefinition constraints and the handling of ambiguity.

4.11 Names and Scoping

An entire IDL file forms a naming scope. In addition, the following kinds of definitions form nested scopes:

- module
- interface
- structure
- union
- operation
- exception

Identifiers for the following kinds of definitions are scoped:

- types
- constants
- enumeration values
- exceptions
- interfaces
- attributes
- operations

An identifier can only be defined once in a scope. However, identifiers can be redefined in nested scopes.

Due to possible restrictions imposed by future language bindings, IDL identifiers are case insensitive—i.e. two identifiers that differ only in the case of their characters are considered redefinitions of one another. However, all references to a definition must use the same case as the defining occurrence. (This allows natural mappings to case-sensitive languages.)

Type names defined in a scope are available for immediate use within that scope. In particular, see §4.7.2 on page 68 on cycles in type definitions.

A name can be used in an unqualified form within a particular scope; it will be resolved by successively searching farther out in enclosing scopes. Once an unqualified name is used in a scope, it cannot be redefined—i.e. if one has used a name defined in an enclosing scope in the current scope, one cannot then redefine a version of the name in the current scope. Such redefinitions yield a compilation error.

A qualified name (one of the form <scoped-name::<identifier>) is resolved by first resolving the qualifier <scoped-name> to a scope S, and then locating the definition of <identifier> within S. The identifier must be directly defined in S or (if S is an interface) inherited into S. The <identifier> is not searched for in enclosing scopes.

When a qualified name begins with "::", the resolution process starts with the smallest enclosing module, and locates subsequent identifiers in the qualified name by the rule described in the previous paragraph.

Every IDL definition in a file has a global name within that file. The global name for a definition is constructed as follows.

Prior to starting to scan a file containing an IDL specification, the name of the current root is initially empty ("") and the name of the current scope is initially empty (""). Whenever a **module** keyword is encountered, the string "::" and the associated identifier are appended to the name of the current root; upon detection of the termination of the **module**, the trailing "::" and identifier are deleted from the name of the current root. Whenever an **interface**, **struct**, **union**, or **exception** keyword is encountered, the string "::" and the associated identifier are appended to the name of the current scope; upon detection of the termination of the **interface**, **struct**, **union**, or **exception**, the trailing "::" and identifier are deleted from the name of the current scope. Additionally, a new, unnamed, scope is entered when the parameters of an operation declaration are processed; this allows the parameter names to duplicate other identifiers; when parameter processing has completed, the unnamed scope is exited.

The global name of an IDL definition is the concatenation of the current root, the current scope, a "::", and the <> which is the local name for that definition.

Inheritance produces shadow copies of the inherited identifiers—i.e., it introduces names into the derived interface, but these names are considered to be semantically "the same" as the original definition. Two shadow copies of the same original (as results from the diamond shape in FIG. 11 on page 61) introduce a single name into the derived interface and don't conflict with each other.

Inheritance introduces multiple global IDL names for the inherited identifiers. Consider the following example:

```
interface A {
      exception E {
            long L;
      };
      void f() raises(E);
};

interface B: A {
      void g() raises(E);
};
```

In this example, the exception is known by the global names **::A::E** and **::B::E**.

Ambiguity can arise in specifications due to the nested naming scopes. For example:

```
interface A {
      typedef string<128> string_t;
};

interface B {
      typedef string<256> string_t;
};

interface C: A, B {
      attribute string_t Title;                  /* AMBIGUOUS!!! */
};
```

The attribute declaration in C is ambiguous, since the compiler does not know which **string_t** is desired. Ambiguous declarations yield compilation errors.

4.12 Differences from C++

The IDL grammar, while attempting to conform to the C++ syntax, is somewhat more restrictive. The current restrictions are as follows:

- A function return type is mandatory.
- A name must be supplied with each formal parameter to an operation declaration.
- A parameter list consisting of the single token **void** is *not* permitted as a synonym for an empty parameter list.
- Tags are required for structures, discriminated unions, and enumerations.
- Integer types cannot be defined as simply int or unsigned; they must be declared explicitly as **short** or **long**.
- **char** cannot be qualified by **signed** or **unsigned** keywords.

4.13 Standard Exceptions

This section presents the standard exceptions defined for the ORB. These exception identifiers may be returned as a result of any operation invocation, regardless of the interface specification. Standard exceptions may not be listed in **raises** expressions.

In order to bound the complexity in handling the standard exceptions, the set of standard exceptions should be kept to a tractable size. This constraint forces the definition of equivalence classes of exceptions rather than enumerating many similar exceptions. For example, an operation invocation can fail at many different points due to the inability to allocate dynamic memory. Rather than enumerate several different exceptions corresponding to the different ways that memory allocation failure causes the exception (during marshalling, unmarshalling, in the client, in the object implementation, allocating network packets, ...), a single exception corresponding to dynamic memory allocation failure is defined. Each standard exception includes a minor code to designate the subcategory of the exception; the assignment of values to the minor codes is left to each ORB implementation.

Each standard exception also includes a **completion_status** code which takes one of the values {YES, NO, MAYBE}. These have the following meanings:

YES	The object implementation has completed processing prior to the exception being raised.
NO	The object implementation was never initiated prior to the exception being raised.
MAYBE	The status of implementation completion is indeterminate.

The standard exceptions are defined below.

```
module StExcep{

    #define ex_body {unsigned long minor; completion_status completed;}

    enum completion_status {YES, NO, MAYBE};
    enum exception_type {NO_EXCEPTION, USER_EXCEPTION, SYSTEM_EXCEPTION};

    exception UNKNOWN          ex_body;   // the unknown exception
    exception BAD_PARAM        ex_body;   // an invalid parameter was passed
    exception NO_MEMORY        ex_body;   // dynamic memory allocation failure
    exception IMP_LIMIT        ex_body;   // vioilated implementation limit
    exception COMM_FAILURE     ex_body;   // communication failure
    exception INV_OBJREF       ex_body;   // invalid object reference
    exception NO_PERMISSION    ex_body;   // no permission for attempted op.
    exception INTERNAL         ex_body;   // ORB internal error
    exception MARSHAL          ex_body;   // error marshalling param/result
    exception INITIALIZE       ex_body;   // ORB initialization failure
    exception NO_IMPLEMENT     ex_body;   // operation implementation unavailable
    exception BAD_TYPECODE     ex_body;   // bad typecode
    exception BAD_OPERATION    ex_body;   // invalid operation
    exception NO_RESOURCES     ex_body;   // insufficient resources for req.
    exception NO_RESPONSE      ex_body;   // response to req. not yet available
    exception PERSIST_STORE    ex_body;   // persistent storage failure
    exception BAD_INV_ORDER    ex_body;   // routine invocations out of order
    exception TRANSIENT        ex_body;   // transient failure - reissue request
    exception FREE_MEM         ex_body;   // cannot free memory
    exception INV_IDENT        ex_body;   // invalid identifier syntax
    exception INV_FLAG         ex_body;   // invalid flag was specified
    exception INTF_REPOS       ex_body;   // error accessing interface repository
    exception CONTEXT          ex_body;   // error processing context object
    exception OBJ_ADAPTER      ex_body;   // failure detected by object adapter
    exception DATA_CONVERSION  ex_body;   // data conversion error
};
```

5 C Language Stub Mapping

The CORBA architecture is independent of the programming language used to construct clients or implementations. In order to use the ORB, it is necessary for programmers to know how to access ORB functionality from their particular programming languages.

This chapter defines the mapping to the C programming language as the first language required by OMG; it is expected that additional language mappings will be defined.

5.1 Requirements for a Language Mapping

All language mappings have approximately the same structure. They must define the means of expressing in the language:

- all IDL basic data types
- all IDL constructed data types
- references to constants defined in IDL
- references to objects defined in IDL
- invocations of operations, including passing parameters and receiving results

- exceptions, including what happens when an operation raises an exception and how the exception parameters are accessed
- access to attributes
- signatures for the operations defined by the ORB, such as the dynamic invocation interface, the object adapters, etc.

A complete language mapping will allow a programmer to have access to all ORB functionality in a way that is convenient for the particular programming language. To support source portability, all ORB implementations must support the same mapping for a particular language.

5.1.1 Basic Data Types

A language mapping must define the means of expressing all of the data types defined in §4.7.1 on page 66. The ORB defines the range of values supported, but the language mapping defines how a programmer sees those values. For example, the C mapping might define TRUE as 1 and FALSE as 0, whereas the LISP mapping might define TRUE as T and FALSE as NIL. The mapping must specify the means to construct and operate on these data types in the programming language.

5.1.2 Constructed Data Types

A language mapping must define the means of expressing the constructed data types defined in §4.7.2 on page 68 through §4.7.4 on page 71. The ORB defines aggregates of basic data types that are supported, but the language mapping defines how a programmer sees those aggregates. For example, the C mapping might define an IDL struct as a C struct, whereas the LISP mapping might define an IDL struct as a list. The mapping must specify the means to construct and operate on these data types in the programming language.

5.1.3 Constants

IDL definitions may contain named constant values that are useful as parameters for certain operations. The language mapping should provide the means to access these constants by name.

5.1.4 Objects

There are two parts of defining the mapping of ORB objects to a particular language. The first specifies how an object is represented in the program and passed as a parameter to operations. The second is how an object is invoked.

The representation of an object reference in a particular language is generally opaque, that is, some language-specific data type is used to represent the object reference, but the program does not interpret the values of that type. The language-specific representation is independent of the ORB representation of an object reference, so that programs are not ORB-dependent. In an object-oriented programming language, it may be convenient to represent an ORB object as a programming language object. Any correspondence between the programming language object types and the IDL types including inheritance, operation names, etc., is up to the language mapping.

There are only three uses that a program can make of an object reference: it may specify it as a parameter to an operation (including receiving it as an output parameter), it can invoke an operation on it, or it can perform an ORB operation (including object adapter operations) on it.

5.1.5 Invocation of Operations

An operation invocation requires the specification of the object to be invoked, the operation to be performed, and the parameters to be supplied. There are a variety of possible mappings, depending to a large extent on the procedure mechanism in the particular language. Some possible choices for language mapping of invocation include: interface-specific stub routines, a single general-purpose routine, a set of calls to construct a parameter list and initiate the operation, or mapping ORB operations to operations on objects defined in an object-oriented programming language.

The mapping must define how parameters are associated with the call, and how the operation name is specified. It is also necessary to specify the effect of the call on the flow of control in the program, including when an operation completes normally and when an exception is raised.

The most natural mapping would be to model a call on an ORB object as the corresponding call in the particular language. However, this may not always be possible for languages where the type system or call mechanism is not powerful enough to handle ORB objects. In this case, multiple calls may be required. For example, in C, it is necessary to have a separate interface for dynamic construction of calls, since C does not permit discovery of new types at runtime. In LISP, however, it may be possible make a language mapping that is the same for objects whether or not they were known at compile time.

In addition to defining how an operation is expressed, it is necessary to specify the storage allocation policy for parameters, for example, what happens to storage of input parameters, and how and where output parameters are allocated. It is also necessary to describe how a return value is handled, for operations that have one.

5.1.6 Exceptions

There are two aspects to the mapping of exceptions into a particular language. First is the means for handling an exception when it occurs, including deciding which exception occurred. If the programming language has a model of exceptions that can accommodate ORB exceptions, that would likely be the most convenient choice; if it does not, some other means must be used, for example, passing additional parameters to the operation that receive the exception status.

It is commonly the case that the programmer associates specific code to handle each kind of exception. It is desirable to make that association as convenient as possible.

When an exception has been raised, it must be possible to access the parameters of the exception. If the language exception mechanism allows for parameters, that mechanism could be used. Otherwise, some other means of obtaining the exception values must be provided.

5.1.7 Attributes

The ORB models attributes as a pair of operations, one to set and one to get the attribute value. The language mapping defines the means of expressing these operations. One reason for distinguishing attributes from pairs of operations is to allow the language mapping to define the most natural way for accessing them. Some possible choices include defining two operations for each attribute, defining two operations that can set or get, respectively, any attribute, defining operations that can set or get groups of attributes, etc.

5.1.8 ORB Interfaces

Most of a language mapping is concerned with how the programmer-defined objects and data are accessed. Programmers who use the ORB must also access some interfaces implemented directly by the ORB, for example, to convert an object reference to a string. A language mapping must also specify how these interfaces appear in the particular programming language.

Various approaches may be taken, including defining a set of library routines, allowing additional ORB-related operations on objects, or defining interfaces that are similar to the language mapping for ordinary objects.

The last approach is called defining pseudo-objects. A pseudo-object has an interface that can (with a few exceptions) be defined in IDL, but is not necessarily implemented as an ORB object.Using stubs a client of a pseudo-object writes calls to it in the same way as if it were an ordinary object. Pseudo-object operations cannot be invoked with the Dynamic Invocation Interface. However, the ORB may recognize such calls as special and handle them directly. One advantage of pseudo-objects is that the interface can be expressed in IDL independent of the particular language mapping, and the programmer can understand how to write calls by knowing the language mapping for the invocations of ordinary objects.

It is not necessary for a language mapping to use the pseudo-object approach. However, this document defines interfaces in subsequent chapters using IDL wherever possible. A language mapping must define how these interfaces are accessed, either by defining them as pseudo-objects and supporting a mapping similar to ordinary objects, by defining language-specific interfaces for them, or in some other way.

5.1.9 Language Stub Mapping

The remainder of this chapter is the C language stub mapping. Subsequent chapters will discuss additional ORB interfaces. Although attempts have been made to separate the mapping-specific details from the ORB-generic concepts, the reader is cautioned that it has not always been possible to separate them. A future revision of this document will restructure this information to clarify which concepts are applicable to any programming language mapping, and which concepts are specific to the C language mapping.

5.2 Scoped Names

The C programmer must always use the global name for a type, constant, exception, or operation. The C global name corresponding to an IDL global name is derived by converting occurrences of "**::**" to "**_**" (an underscore) and eliminating the leading underscore.

Consider the following example:

```
typedef string<256> filename_t;
interface example0 {
    enum color {red, green, blue};
    union bar switch (enum foo {room, bell}) { ... };
    •••
};
```

Code to use this interface would look as follows:

```
#include "example0.h"                                          /* C */

filename_t FN;
example0_color C = example0_red;
example0_bar myUnion;

switch (myUnion._d) {
   case example0_bar_room: • • •
   case example0_bar_bell: • • •
};
```

Note that the use of underscores to replace the "::" separators can lead to ambiguity if the IDL specification contains identifiers with underscores in them. Consider the following example:

```
typedef long foo_bar;
interface foo {
    typedef short bar;           /* A legal IDL statement, but ambiguous in C */
    •••
};
```

Due to such ambiguities, it is advisable to avoid the indiscriminate use of underscores in identifiers.

5.3 Mapping for Interfaces

All interfaces must be defined at global scope (*no* nested interfaces). The mapping for an interface declaration is as follows:

```
interface example1 {
    long op1(in long arg1);
};
```

The preceding example generates the following C declarations[1]:

```
typedef Object example1;                                          /* C */
extern long example1_op1(example1 o, Environment *ev, long arg1);
```

All object references (actually typed interface references to an object) are of the well-known, opaque type **`Object`**. The representation of Object is a pointer. To permit the programmer to decorate a program with typed references, a type with the name of the interface is defined to be an **`Object`**. The literal **`OBJECT_NIL`** is legal wherever an **`Object`** may be used; it is guaranteed to pass the **is_nil** operation defined in §8.2.3 on page 146.

IDL permits specifications in which arguments, return results, or components of constructed types may be interface references. Consider the following example:

```
#include "example1.idl"

interface example2 {
    example1 op2();
};
```

1. §5.14 on page 95 describes the additional arguments added to an operation in the C mapping.

This is equivalent to the following C declaration:

```
#include "example1.h"                                          /* C */

typedef Object example2;
extern example1 example2_op2(example2 o, Environment *ev);
```

A C fragment for invoking such an operation is as follows:

```
#include "example2.h"                                          /* C */

example1 ex1;
example2 ex2;
Environment ev;

/* code for binding ex2 */

ex1 = example2_op2(ex2, &ev);
```

5.4 Inheritance and Operation Names

IDL permits the specification of interfaces that inherit operations from other interfaces. Consider the following example:

```
interface example3 : example1 {
   void op3(in long arg3, out long arg4);
};
```

This is equivalent to the following C declarations:

```
typedef Object example3;                                       /* C */
extern long example3_op1(example3 o, Environment *ev, long arg1);
extern void example3_op3(
   example3 o,
   Environment *ev,
   long arg3,
   long *arg4
);
```

As a result, an object written in C can access **op1** as if it was directly declared in **example3**. Of course, the programmer could also invoke **example1_op1** on an **Object** of type **example3**; the virtual nature of operations in interface definitions will cause invocations of either function to cause the same method to be invoked.

5.5 Mapping for Attributes

The mapping for attributes is best explained through a couple of examples. Consider the following specification:

```
interface foo {
    struct position_t {
        float x, y;
    };

    attribute float radius;
    readonly attribute position_t position;
};
```

This is exactly equivalent to the following illegal IDL specification:

```
interface foo {
    struct position_t {
        float x, y;
    };

    float         _get_radius();
    void          _set_radius(in float r);
    position_t    _get_position();
};
```

This latter specification is illegal since IDL identifiers are not permitted to start with the underscore (_) character.

The language mapping for attributes then becomes the language mapping for these equivalent operations. More specifically, the function signatures generated for the above operations are as follows:

```
typedef struct foo_position_t {                              /* C */
   float x, y;
} foo_position_t;

extern float foo__get_radius(foo o, Environment *ev);
extern void foo__set_radius(foo o, Environment *ev, float r);
extern foo_position_t foo__get_position(foo o, Environment *ev);
```

Note that two underscore characters (__) separate the name of the interface from the words "**get**" or "**set**" in the names of the functions.

If the "**set**" accessor function fails to set the attribute value, the method should return one of the standard exceptions defined in §4.13 on page 79.

5.6 Mapping for Constants

Constant identifiers can be referenced at any point in the user's code where a literal of that type is legal. In C, these constants are **`#define`**d.

The fact that constants are **`#define`**d may lead to ambiguities in code. All names which are mandated by the mappings for any of the structured types below start with an underscore.

5.7 Mapping for Basic Data Types

The basic data types have the mappings shown in TBL. 11 on page 89. Note that the **`unsigned char`** mapping of the IDL **boolean** types defines only the values 1 (TRUE) and 0 (FALSE); other values produce undefined behavior.

TBL. 11 Data Type Mappings

IDL	C
short	short
long	long
unsigned short	unsigned short
unsigned long	unsigned long
float	float
double	double
char	char
boolean	unsigned char
octet	unsigned char
enum	unsigned long
any	typedef struct any { TypeCode _type; void *_value; } any;

Each enumerator in an **enum** is **`#define`**d with an appropriate **`unsigned long`** value conforming to the ordering constraints described in §4.7.2.3 on page 70.

TypeCodes are described in §7.6 on page 138. The **`_value`** member for an **any** is a pointer to the actual value of the datum.

5.8 Mapping for Structure Types

IDL structures map directly onto C **`struct`**s. Note that all IDL types that map to C **`struct`**s may potentially include padding.

5.9 Mapping for Union Types

IDL discriminated unions are mapped onto C **`struct`**s. Consider the following IDL declaration:

```
union Foo switch (long) {
    case 1: long x;
    case 2: float y;
    default: char z;
};
```

This is equivalent to the following **struct** in C:

```
typedef struct {                                        /* C */
   long _d;
   union {
      long x;
      float y;
      char z;
   } _u;
} Foo;
```

The discriminator in the struct is always referred to as **`_d`**; the union in the struct is always referred to as **`_u`.**

Reference to union elements is as in normal C:

```
Foo *v;                                                 /* C */

/* make a call that returns a pointer to a Foo in v */

switch(v->_d) {
   case 1:  printf("x = %ld\n", v->_u.x); break;
   case 2:  printf("y = %f\n", v->_u.y); break;
   default: printf("z = %c\n", v->_u.z); break;
}
```

5.10 Mapping for Sequence Types

The IDL data type **sequence** permits passing of unbounded arrays between objects. Consider the following IDL declaration:

```
typedef sequence<long,10> vec10;
```

In C, this is converted to:

```
typedef struct {                                        /* C */
   unsigned long _maximum;
   unsigned long _length;
   long *_buffer;
} vec10;
```

An instance of this type is declared as follows:

```
vec10 x = {10L, 0L, (long *)NULL);                      /* C */
```

Prior to passing **x** as an **in** parameter, the programmer must set the **_buffer** member to point to a **long** array of 10 elements, and must set the **_length** member to the actual number of elements to transmit.

Prior to passing **&x** as an **out** parameter (or receiving a **vec10** as the function return), the programmer does nothing. The client stub will allocate storage for the returned buffer; for bounded sequences, it allocates a buffer of the specified size, while for unbounded sequences, it allocates a buffer big enough to hold what was returned by the object. Upon successful return from the invocation, the **_maximum** member will contain the size of the allocated array, the **_buffer** member will point at the allocated storage, and the **_length** member will contain the number of values that were returned in the **_buffer** member. The client is responsible for freeing the allocated storage using **ORBfree()**. The current contents of the **_buffer** member is overwritten with a pointer to the storage allocated by the stub.

Prior to passing **&x** as an **inout** parameter, the programmer must set the **_buffer** member to point to a **long** array of 10 elements. For an unbounded sequence, the programmer *must* set the **_maximum** member to the actual size of the array. The **_length** member must be set to the actual number of elements to transmit. Upon successful return from the invocation, the **_length** member will contain the number of values that were copied into the buffer pointed to by the **_buffer** member. The number of values returned is constrained by the value of the **_maximum** member.

For bounded sequences, it is an error to set the **_length** or **_maximum** member to a value larger than the specified bound.

Two sequence types are the same type if their sequence element type and size arguments are identical. For example,

```
const long SIZE = 25;
typedef long seqtype;

typedef sequence<long, SIZE> s1;
typedef sequence<long, 25> s2;
typedef sequence<seqtype, SIZE> s3;
typedef sequence<seqtype, 25> s4;
```

declares **s1**, **s2**, **s3**, and **s4** to be of the same type.

The IDL type

```
sequence<type,size>
```

maps to

```
#ifndef _IDL_SEQUENCE_type_defined                          /* C */
#define _IDL_SEQUENCE_type_defined
typedef struct {
   unsigned long _maximum;
   unsigned long _length;
   type *_buffer;
} _IDL_SEQUENCE_type;
#endif   /* _IDL_SEQUENCE_type_defined */
```

The **ifdef**’s are needed to prevent duplicate definition where the same type is used more than once. The type name used in the C mapping is the type name of the effective type, e.g. in

```
typedef long FRED;                                          /* C */
typedef sequence<FRED,10> FredSeq;
```

the sequence is mapped onto **struct { ... } _IDL_SEQUENCE_long;**

If the **type** in

```
sequence<type,size>
```

consists of more than one identifier (e.g. unsigned long), then the generated type name consists of the string “_IDL_SEQUENCE_” concatenated to the string consisting of the concatenation of each identifier separated by underscores (e.g. “unsigned_long”).

If the **type** is a **string**, the string “string” is used to generate the type name. If the **type** is a **sequence**, the string “sequence” is used to generate the type name, recursively. For example

```
sequence<sequence<long>>
```

generates a type of

```
_IDL_SEQUENCE_sequence_long
```

These generated type names may be used to declare instances of a sequence type.

5.11 Mapping for Strings

IDL strings are mapped to 0-byte terminated character arrays; i.e. the length of the string is encoded in the character array itself through the placement of the 0-byte. Note that the storage for C strings is one byte longer than the stated IDL bound. Consider the following IDL declarations:

```
typedef string<10> sten;
typedef string sinf;
```

In C, this is converted to:

```
typedef char *sten;                                    /* C */
typedef char *sinf;
```

Instances of these types are declared as follows:

```
sten s1 = NULL;                                        /* C */
sinf s2 = NULL;
```

Two string types are the same type if their size arguments are identical. For example,

```
const long SIZE = 25;                                  /* C */

typedef string<SIZE> sx;
typedef string<25> sy;
```

declares **sx** and **sy** to be of the same type.

Prior to passing **s1** or **s2** as an **in** parameter, the programmer must assign the address of a character buffer containing a 0-byte terminated string to the variable.

Prior to passing **&s1** or **&s2** as an **out** parameter (or receiving an **sten** or **sinf** as the return result), the programmer does nothing. The client stub will allocate storage for the returned buffer; for bounded strings, it allocates a buffer of the specified size, while for unbounded strings, it allocates a buffer big enough to hold the returned string. Upon successful return from the invocation, the character pointer will contain the address of the allocated buffer. The client is responsible for freeing the allocated storage using

`ORBfree()`. If the pointer is non-NULL when a call is made, it is overwritten with a pointer to the storage allocated by the stub.

Prior to passing **`&s1`** or **`&s2`** as an **inout** parameter, the programmer must assign the address of a character buffer containing a 0-byte terminated array to the variable. Upon successful return from the invocation, the returned 0-byte terminated array is copied into the same buffer. If it was a bounded string, then the size of the returned string is limited by the declared size of the string type; if it was an unbounded string, then the size of the returned string is limited by the size of the string passed as input. Due to this restriction, use of **inout** string parameters is deprecated.

5.12 Mapping for Arrays

IDL arrays map directly to C arrays. All array indices run from 0 to <**`size`** - 1>.

If the return result to an operation is an array, the array storage is dynamically allocated by the stub; a pointer to the first element of the dynamically allocated array is returned as the value of the client stub function. When the data is no longer needed, it is the programmer's responsibility to return the dynamically allocated storage by calling **`ORBfree()`**.

5.13 Mapping for Exception Types

Each defined exception type is defined as a struct tag and a typedef with the C global name for the exception. An identifier for the exception, in string literal form, is also **`#define`**d. For example:

```
exception foo {
    long dummy;
};
```

yields the following C declarations:

```
typedef struct foo {                                    /* C */
   long dummy;
} foo;
#define ex_foo <unique identifier for exception>
```

The identifier for the exception uniquely identifies this exception type. For example, it could be the Interface Repository identifier for the exception (see §7.5.10 on page 138).

5.14 Implicit Arguments to Operations

From the point of view of the C programmer, all operations declared in an interface have additional leading parameters preceding the operation-specific parameters:

1. The first parameter to each operation is an **`Object`** input parameter; this parameter designates the object to process the request.
2. The second parameter to each operation is an (**`Environment *`**) output parameter; this parameter permits the return of exception information.
3. If an operation in an IDL specification has a context specification, then a **`Context`** input parameter follows the (**`Environment *`**) parameter and precedes any operation-specific arguments.

As described above, the **`Object`** type is an opaque type. The **`Environment`** type is partially opaque; §5.19 on page 99 provides a description of the non-opaque portion of the exception structure and an example of how to handle exceptions in client code. The **`Context`** type is opaque; see Chapter 6 for more information on how to create and manipulate context objects.

5.15 Interpretation of Functions with Empty Argument Lists

A function declared with an empty argument list is defined to take *no* operation-specific arguments.

5.16 Argument Passing Considerations

For all IDL types (except arrays), if the IDL signature specifies that an argument is an **out** or **inout** parameter, then the caller must always pass the address of a variable of that type (or the value of a pointer to that type); the callee must dereference the parameter to get to the type. For arrays, the caller must pass the address of the first element of the array.

For **in** parameters, the value of the parameter must be passed for all of the basic types, enumeration types, and object references. For arrays, the address of the first element of the array must be passed. For all other structured types, the address of a variable of that type must be passed.

Consider the following IDL specification:

```
interface foo {
    typedef long Vector[25];

    void bar(out Vector x, out long y);
};
```

Client code for invoking the **bar** operation would look like:

```
foo object;                                               /* C */
foo_Vector x;
long y;
Environment ev;

/* code to bind object to instance of foo */

foo_bar(object, &ev, x, &y);
```

For **out** parameters of type unbounded **string** or unbounded **sequence**, the ORB will allocate storage for the output value. The client may use and retain that storage indefinitely, and must indicate when the value is no longer needed by calling the procedure **ORBfree**, whose signature is:

```
extern void ORBfree (void *storage);                      /* C */
```

The parameter to **ORBfree()** is the pointer used to return the **out** parameter. **ORBfree()** releases the ORB-allocated storage occupied by the **out** parameter, including storage indirectly referenced, such as in the case of a sequence of strings. If a client does not call **ORBfree()** before reusing the pointers that reference the **out** parameters, that storage might be wasted.

5.17 Return Result Passing Considerations

When an operation is defined to return a non-void return result, the following rules hold:

1. If the return result is one of the types **float**, **double**, **long**, **short**, **unsigned long**, **unsigned short**, **char**, **boolean**, **octet**, **Object**, or an **enumeration**, then the value is returned as the operation result.
2. If the return result is one of the types **struct**, **union**, **sequence**, or **any**, then the value of the C struct representing that type is returned as the operation result.
3. If the return result is of type **string**, then a pointer to the first character of the string is returned as the operation result.
4. If the return result is of type **array**, then a pointer to the first element of the array is returned as the operation result.

Consider the following interface:

```
interface X {
    struct y {
        long a;
        float b;
    };

    long op1();
    y op2();
}
```

The following C declarations ensue from processing the specification:

```
typedef Object X;                                          /* C */
typedef struct X_y {
   long a;
   float b;
} X_y;

extern long X_op1(X object, Environment *ev);
extern X_y X_op2(X object, Environment *ev);
```

For operation results of type unbounded **string**, unbounded **sequence**, or **array**, the ORB will allocate storage for the return value. The client may use and retain that storage indefinitely, and must indicate when the value is no longer needed by calling the procedure **`ORBfree()`** described in §5.16 on page 95.

5.18 Summary of Argument/Result Passing

TBL. 12 on page 97 summarizes what a client passes as an argument to a stub and receives as a result.

TBL. 12 Argument and Result Passing

Data Type	Pass In	Pass Out/Inout	Return Result
short	value	addr of variable to hold value	receive value
long	value	addr of variable to hold value	receive value
unsigned short	value	addr of variable to hold value	receive value
unsigned long	value	addr of variable to hold value	receive value
float	value	addr of variable to hold value	receive value
double	value	addr of variable to hold value	receive value
boolean	value	addr of variable to hold value	receive value

TBL. 12 Argument and Result Passing (Continued)

Data Type	Pass In	Pass Out/Inout	Return Result
char	value	addr of variable to hold value	receive value
octet	value	addr of variable to hold value	receive value
enumeration	value	addr of variable to hold value	receive value
object reference	Object value	addr of variable to hold Object	receive value of Object
struct	addr of struct	addr of variable to hold struct	receive value of struct
union	addr of struct	addr of variable to hold struct	receive value of struct
string	addr of 1st char	addr of (char *) variable	receive char *
sequence	addr of seq. struct	addr of seq. struct	receive value of seq. struct
array	addr of 1st elem	addr of 1st elem	receive addr of 1st elem

A client is responsible for providing storage for all arguments passed as **in** arguments. A client is responsible for providing storage for all **out** arguments and return results except in the cases noted in TBL. 13 on page 98 and TBL. 14 on page 99.

TBL. 13 Client Argument Storage Responsibilities

Type	Inout Param	Out Param	Return Result
short	1	1	1
long	1	1	1
unsigned short	1	1	1
unsigned long	1	1	1
float	1	1	1
double	1	1	1
boolean	1	1	1
char	1	1	1
octet	1	1	1
enumeration	1	1	1
object reference	2	2	2
struct	1	1	1
union	1	1	1
string	1	3	3
sequence	1	4	4
array	1	1	3
any	4	4	4

TBL. 14 Argument Passing Cases

Case[1]	Description
1	The client is responsible for providing storage and managing release of the storage. That is, the system allocates storage which must be freed using ORBfree() for the following types and directions: string/out, string/return, sequence/out, sequence/return, array/return, any/inout, any/out, any/return. For inout strings and sequences, the out result is constrained by the size of the type on input.
2	The client is responsible for providing the storage used to contain the object reference. When the reference is no longer needed, the client uses the release function described in §8.2.2 on page 145 to release the storage associated with the reference.
3	The ORB provides the storage for these returned parameters and results. The client is responsible for releasing the storage using ORBfree().
4	The client provides the storage for the structure which contains the description of the sequence or any and the client manages release of the storage and the descriptor. The ORB provides storage for the values returned and puts the pointers to this storage in the descriptor structures. The client is responsible for releasing this storage using ORBfree().

1. As listed in TBL. 13 on page 98.

5.19 Handling Exceptions

The **Environment** type is partially opaque; the C declaration contains at least the following:

```
typedef struct Environment {                          /* C */
   exception_type _major;
   ...
} Environment;
```

Upon return from an invocation, the **_major** field indicates whether the invocation terminated successfully; **_major** can have one of the values **NO_EXCEPTION**, **USER_EXCEPTION**, or **SYSTEM_EXCEPTION**; if the value is one of the latter two, then any exception parameters signalled by the object can be accessed.

Three functions are defined on an **Environment** structure for accessing exception information; their signatures are:

```
extern char *exception_id(Environment *ev);           /* C */
extern void *exception_value(Environment *ev);
extern void exception_free(Environment *ev);
```

`exception_id()` returns a pointer to the character string identifying the exception. If invoked on an **`Environment`** which identifies a non-exception, (**`_major==NO_EXCEPTION`**) a NULL is returned.

`exception_value()` returns a pointer to the structure corresponding to this exception. If invoked on an **`Environment`** which identifies a non-exception or an exception for which there is no associated information, a NULL is returned.

`exception_free()` returns any storage which was allocated in the construction of the **`Environment`**. It is permissible to invoke **`exception_free()`** regardless of the value of the **`_major`** field.

Consider the following example:

```
interface exampleX {
    exception BadCall {
    string<80> reason;
    };

    void op() raises(BadCall);
};
```

This interface defines a single operation which returns no results and can raise a **BadCall** exception. The following user code shows how to invoke the operation and recover from an exception:

```
#include "exampleX.h"                                          /* C */

Environment ev;
exampleX obj;
exampleX_BadCall *bc;

/*
 * some code to initialize obj to a reference to an object supporting
 * the exampleX interface
 */

exampleX_op(obj, &ev);
switch(ev._major) {
case NO_EXCEPTION:/* successful outcome*/
   /* process out and inout arguments */
   break;
case USER_EXCEPTION:/* a user-defined exception */
   if (strcmp(ex_exampleX_BadCall, exception_id(&ev)) == 0) {
      bc = (exampleX_BadCall *)exception_value(&ev);
      fprintf(stderr, "exampleX_op() failed - reason: %s\n",
         bc->reason);
      else {   /* should never get here ... */
         fprintf(stderr, "unknown user-defined exception - %s\n",
            exception_id(&ev));
      }
      break;
default: /* standard exception */
   /*
    * exception_id() can be used to determine which particular
    * standard exception was raised; the minor member of the struct
    * associated with the exception (as yielded by exception_value())
    * may provide additional system-specific information about the
    * exception
    */
   break;
}
exception_free(&ev);/* free any storage associated with exception */
```

5.20 Method Routine Signatures

The signatures of the methods used to implement an object depend not only on the language binding, but also on the choice of object adapter. Different object adapters may provide additional parameters to access object adapter-specific features.

Most object adapters are likely to provide method signatures that are similar in most respects to those of the client stubs. In particular, the mapping for the operation parameters expressed in IDL should be the same as for the client side.

See §9.3 on page 156 for the description of method signatures for implementations using the Basic Object Adapter.

5.21 Include Files

Multiple interfaces may be defined in a single source file. By convention, each interface is stored in a separate source file. All IDL compilers will, by default, generate a header file named **Foo.h** from **Foo.idl**. This file should be **#include**d by clients and implementations of the interfaces defined in **Foo.idl**.

Inclusion of **Foo.h** is sufficient to define all global names associated with the interfaces in **Foo.idl** and any interfaces from which they are derived.

5.22 Pseudo-objects

In the C language mapping, there are several interfaces that are defined as pseudo-objects; TBL. 18 on page 172 lists the pseudo-objects. A client makes calls on a pseudo-object in the same way as an ordinary ORB object. However, the ORB may implement the pseudo-object directly, and there are restrictions on what a client may do with a pseudo-object.

The ORB itself is a pseudo-object with the following partial definition (see Chapter 8 for the complete definition):

```
interface ORB {
    string      object_to_string (in Object obj);
    Object      string_to_object (in string str);
};
```

This means that a C programmer may convert an object reference into its string form by calling:

```
Environment ev;                                      /* C */
char *str = ORB_object_to_string (orbobj, &ev, obj);
```

just as if the ORB were an ordinary object. The C library contains the routine **ORB_object_to_string**, and it it does not do a real invocation. The **orbobj** is an object reference that specifies which ORB is of interest, since it is possible to choose which ORB should be used to convert an object reference to a string (see Chapter 8 for details on this specific operation).

Although operations on pseudo-objects are invoked in the usual way defined by the C language mapping, there are restrictions on them. In general, a pseudo-object cannot be specified as a parameter to an operation on an ordinary object. Pseudo-objects are also not accessible using the dynamic invocation interface, and do not have definitions in the interface repository.

Operations on pseudo-objects may take parameters that are not permitted in operations on ordinary objects. For example, the **set_exception** operation on the Basic Object Adapter pseudo-object takes a C `(void *)` to specify the exception parameters (see §9.3.2 on page 157 for details). Generally, these parameters will be language-mapping specific.

Because the programmer uses pseudo-objects in the same way as ordinary objects, some ORB implementations may choose to implement some pseudo-objects as ordinary objects. For example, assuming it could be efficient enough, a context object might be implemented as an ordinary object.

6 Dynamic Invocation Interface

6.1 Overview

The ORB dynamic invocation interface allows dynamic creation and invocation of requests to objects. A client using this interface to send a request to an object obtains the same semantics as a client using the operation stub generated from the type specification.

A request is comprised of an object reference, an operation, and a list of parameters. The ORB applies the implementation-hiding (encapsulation) principle to requests.

In the dynamic invocation interface, parameters in a request are supplied as elements of a list. Each element is an instance of a **NamedValue** (see §6.1.1 on page 106). Each parameter is passed in its native data form.

Parameters supplied to a request may be subject to run-time type checking upon request invocation. Parameters must be supplied in the same order as the parameters defined for the operation in the Interface Repository.

6.1.1 Common Data Structures

The type **NamedValue** is a well-known datatype in IDL. It can be used either as a parameter type directly or as a mechanism for describing arguments to a request. The type NVList is a pseudo-object useful for constructing parameter lists. The types are described in IDL and C, respectively, as:

```
typedef unsigned long Flags;

struct NamedValue {
    Identifier      name;       // argument name
    any             argument;   // argument
    long            len;        // length/count of argument value
    Flags           arg_modes;  // argument mode flags
};

NamedValue * NVList;                                    /* C */
```

The **NamedValue** and **NVList** structures are used in the request operations to describe arguments and return values. They are also used in the context object routines to pass lists of property names and values. Despite the above declaration for **NVList**, the **NVList** structure is partially opaque and may only be created by using the ORB **create_list** operation.

A named value includes an argument name, argument value (as an **any**), length of the argument, and a set of argument mode flags. When named value structures are used to describe arguments to a request, the names are the argument identifiers specified in the IDL definition for a specific operation.

As described in §5.7 on page 89, an **any** consists of a **TypeCode** and a pointer to the data value. The TypeCode is a well-known opaque type that can encode a description of any type specifiable in IDL. A full description of Typecodes is given in §7.6 on page 138.

For most datatypes, **len** is the actual number of bytes that the value occupies. For object references, **len** is 1. TBL. 15 on page 106 shows the length of data values for the C language binding. The behavior of a NamedValue is undefined if the **len** value is inconsistent with the TypeCode.

TBL. 15 C Type Lengths

Data type: X	Length (X)
short	sizeof (short)
unsigned short	sizeof (unsigned short)
long	sizeof (long)

TBL. 15 C Type Lengths (Continued)

Data type: X	Length (X)
unsigned long	sizeof (unsigned long)
float	sizeof (float)
double	sizeof (double)
char	sizeof (char)
boolean	sizeof (char)
octet	sizeof (char)
string	strlen (string) /* does NOT include '\0' byte! */
enum E {};	sizeof (unsigned long)
union U { } ;	sizeof (U)
struct S { };	sizeof (S)
Object	1
array N of type T1	Length (T1) * N
sequence V of type T2	Length (T2) * V /* V is the actual, dynamic, number of elements */

The **arg_modes** field is defined as a bitmask (long) and may contain the following flag values:

ARG_IN the associated value is an input only argument

ARG_OUT the associated value is an output only argument

ARG_INOUT the associated value is an in/out argument

These flag values identify the parameter passing mode for arguments. Additional flag values have specific meanings for request and list routines, and are documented with their associated routines.

All other bits are reserved. The high-order 16 bits are reserved for implementation-specific flags.

6.1.2 Memory Usage

The values for output argument data types that are unbounded strings or unbounded sequences are returned as pointers to dynamically allocated memory (see TBL. 13 on page 98). In order to facilitate the freeing of all "out-arg memory", the request routines provide a mechanism for grouping, or keeping track of, this memory. If so specified, out-arg memory is associated with the argument list passed to the create request routine. When the list is deleted the associated out-arg memory will automatically be freed.

If the programmer chooses not to associate out-arg memory with an argument list, the programmer is responsible for freeing each out parameter using **`ORBfree()`** (see §5.16 on page 95).

6.1.3 Return Statuses and Exceptions

In the Dynamic Invocation Interface as specified, many routines return an **ORBStatus** result, which is intended as a status code. **ORBStatus** is defined as:

```
typedef unsigned long ORBStatus;
```

Conforming CORBA implementations are *not* required to return this status code; instead, the definition

```
typedef void ORBStatus;
```

is a conforming implementation (in which case no status code result is returned, except in the usual **inout Environment** argument). Implementations are required to specify which **ORBStatus** behavior is supported.

6.2 Request Routines

The request routines are defined in terms of the Request pseudo-object. The Request routines use the **`NVList`** definition defined above.

```
interface Request {                                                    // PIDL

    ORBStatus add_arg (
        in Identifier       name,           // argument name
        in TypeCode         arg_type,       // argument datatype
        in void             * value,        // argument value to be added
        in long             len,            // length/count of argument value
        in Flags            arg_flags       // argument flags
    );
    ORBStatus invoke (
        in Flags            invoke_flags    // invocation flags
    );
    ORBStatus delete ();
    ORBStatus send (
        in Flags            invoke_flags// invocation flags
    );
    ORBStatus get_response (
        in Flags            response_flags  // response flags
    );
};
```

6.2.1 create_request

Because it creates a pseudo-object, this operation is defined in the Object interface (see §8.2 on page 145 for the complete interface definition). The **create_request** operation is performed on the Object which is to be invoked.

```
ORBStatus create_request (                                                    // PIDL
    in Context          ctx,            // context object for operation
    in Identifier       operation,      // intended operation on object
    in NVList           arg_list,       // args to operation
    inout NamedValue    result,         // operation result
    out Request         request,        // newly created request
    in Flags            req_flags       // request flags
);
```

This operation creates an ORB request. The actual invocation occurs by calling **invoke** or by using the **send / get_response** calls.

The operation name specified on **create_request** is the same operation identifier that is specified in the IDL definition for this operation.

The **arg_list**, if specified, contains a list of arguments (input, output, and/or input/output) which become associated with the request. If **arg_list** is omitted (specified as NULL), the arguments (if any) must be specified using the **add_arg** call below.

Arguments may be associated with a request by passing in an argument list or by using repetitive calls to **add_arg**. One mechanism or the other may be used for supplying arguments to a given request; a mixture of the two approaches is not supported.

If specified, the **arg_list** becomes associated with the request; until the **invoke** call has completed (or the request has been deleted), the ORB assumes that **arg_list** (and any values it points to) remains unchanged.

When specifying an argument list, the **value** and **len** for each argument must be specified. An argument's datatype, name, and usage flags (i.e., in, out, inout) may also be specified; if so indicated, arguments are validated for datatype, order, name, and usage correctness against the set of arguments expected for the indicated operation.

An implementation of the request services may relax the order constraint (and allow arguments to be specified out of order) by doing ordering based upon argument name.

The context properties associated with the operation are passed to the object implementation. The object implementation may not modify the context information passed to it.

The operation result is placed in the **result** argument after the invocation completes.

The **req_flags** argument is defined as a bitmask (**long**) that may contain the following flag values:

OUT_LIST_MEMORY Indicates that any out-arg memory is associated with the argument list (**`NVList`**).

Setting the OUT_LIST_MEMORY flag controls the memory allocation mechanism for out-arg memory (output arguments, for which memory is dynamically allocated). If OUT_LIST_MEMORY is specified, an argument list must also have been specified on the **create_request** call. When output arguments of this type are allocated, they are associated with the list structure. When the list structure is freed (see below), any associated out-arg memory is also freed.

If OUT_LIST_MEMORY is *not* specified, then each piece of out-arg memory remains available until the programmer explicitly frees it with the **`ORBfree()`** procedure provided by the language mapping (see §5.16 on page 95).

The operation name is a string that conforms to the IDL rules for naming identifiers.

6.2.2 add_arg

```
ORBStatus add_arg (                                                          // PIDL
      in Identifier          name,          // argument name
      in TypeCode            arg_type,      // argument datatype
      in void                * value,       // argument value to be added
      in long                len,           // length/count of argument value
      in Flags               arg_flags      // argument flags
);
```

add_arg incrementally adds arguments to the request.

For each argument, minimally its **value** and **len** must be specified. An argument's datatype, name, and usage flags (i.e in, out, inout) may also be specified. If so indicated, arguments are validated for datatype, order, name, and usage correctness against the set of arguments expected for the indicated operation.

An implementation of the request services may relax the order constraint (and allow arguments to be specified out of order) by doing ordering based upon argument name.

The arguments added to the request become associated with the request and are assumed to be unchanged until the invoke has completed (or the request has been deleted).

Arguments may be associated with a request by specifying them on the **create_request** call or by adding them via calls to **add_arg**. Using both methods for specifying arguments, for the same request, is not currently supported.

In addition to the argument modes defined in §6.1.1 on page 106, **arg_flags** may also take the flag value: IN_COPY_VALUE. The argument passing flags defined in §6.1.1 on page 106 may be used here to indicate the intended parameter passing mode of an argument.

If the IN_COPY_VALUE flag is set, a copy of the argument value is made and used instead. This flag is ignored for inout and out arguments.

6.2.3 invoke

```
ORBStatus invoke (                                              // PIDL
    in Flags            invoke_flags        // invocation flags
);
```

This operation calls the ORB, which performs method resolution and invokes an appropriate method. If the method returns successfully, its result is placed in the **result** argument specified on **create_request**.

6.2.4 delete

```
ORBStatus delete ( );                                           // PIDL
```

This operation deletes the request. Any memory associated with the request (i.e. by using the **IN_COPY_VALUE** flag) is also freed.

6.3 Deferred Synchronous Routines

6.3.1 send

```
ORBStatus send (                                                // PIDL
    in Flags            invoke_flags          // invocation flags
);
```

send initiates an operation according to the information in the Request. Unlike **invoke**, **send** returns control to the caller without waiting for the operation to finish. To determine when the operation is done, the caller must use the **get_response** or **get_next_response** operations described below. The out parameters and return value must not be used until the operation is done.

Although it is possible for some standard exceptions to be raised by the **send** operation, there is no guarantee that all possible errors will be detected. For example, if the object reference is not valid, **send** might detect it and raise an exception, or might return before the object reference is validated, in which case the exception will be raised when **get_response** is called.

If the operation is defined to be **oneway** or if INV_NO_RESPONSE is specified, then **get_response** does not need to be called. In such cases, some errors might go unreported, since if they are not detected before **send** returns there is no way to inform the caller of the error.

The following invocation flags are currently defined for **send**:

INV_NO_RESPONSE	Indicates that the invoker does not intend to wait for a response, nor does it expect any of the output arguments (in/out and out) to be updated. This option may be specified even if the operation has not been defined to be **oneway**.

6.3.2 send_multiple_requests

```
ORBStatus send_multiple_requests (                              /* C */
   Request        reqs[],            /* array of Requests */
   Environment    *env,
   long           count,             /* number of Requests */
   Flags          invoke_flags
);
```

send_multiple_requests initiates more than one request in parallel. Like **send**, **send_multiple_requests** returns to the caller without waiting for the operations to finish. To determine when each operation is done, the caller must use the **get_response** or **get_next_response** operations described below.

The degree of parallelism in the initiation and execution of the requests is system dependent. There are no guarantees about the order in which the requests are initiated. If INV_TERM_ON_ERR is specified, and the ORB detects an error initiating one of the requests, it will not initiate any further requests from this list. If INV_NO_RESPONSE is specified, it applies to all of the requests in the list.

The following invocation flags are currently defined for **send_multiple_requests**:

INV_NO_RESPONSE	Indicates that the invoker does not intend to wait for a response, nor does it expect any of the output arguments (inout and out) to be updated. This option may be specified even if the operation has not been defined to be **oneway**.
INV_TERM_ON_ERR	If one of the requests causes an error, the remaining requests are not sent.

6.3.3 get_response

```
ORBStatus get_response (                                                    // PIDL
    in Flags                response_flags              // response flags
```

get_response determines whether a request has completed. If **get_response** indicates that the operation is done, the out parameters and return values defined in the Request are valid, and they may be treated as if the Request **invoke** operation had been used to perform the request.

If the RESP_NO_WAIT flag is set, **get_response** returns immediately even if the request is still in progress. Otherwise, **get_response** waits until the request is done before returning.

The following response flags are defined for **get_response**:

RESP_NO_WAIT Indicates that the caller does not want to wait for a response.

6.3.4 get_next_response

```
ORBStatus get_next_response (                                    /* C */
   Environment    *env,
   Flags          response_flags,
   Request        *req
);
```

get_next_response returns the next request that completes. Despite the name, there is no guaranteed ordering among the completed requests, so the order in which they are returned from successive **get_next_response** calls is not necessarily related to the order in which they finished.

If the RESP_NO_WAIT flag is set, and there are no completed requests pending, then **get_next_response** returns immediately. Otherwise, **get_next_response** waits until some request finishes.

The following response flags are defined for **get_next_response**:

RESP_NO_WAIT Indicates that the caller does not want to wait for a response.

6.4 List Routines

The list routines use the named-value structure defined above.

The list operations that create **NVList** objects are defined in the ORB interface described in Chapter 8, but are described in this section. The **NVList** interface is shown below.

```
interface NVList {                                                        // PIDL
    ORBStatus add_item (
        in Identifier    item_name,          // name of item
        in TypeCode      item_type,          // item datatype
        in void          *value,             // item value
        in long          value_len,          // length of item value
        in Flags         item_flags          // item flags
    );
    ORBStatus free ( );
    ORBStatus free_memory ( );
    ORBStatus get_count (
        out long         count               // number of entries in the list
    );

};
```

6.4.1 create_list

This operation, which creates a pseudo-object, is defined in the ORB interface and excerpted below.

```
ORBStatus create_list (                                                   //PIDL
    in long          count,       // number of items to allocate for list
    out NVList       new_list     // newly created list
);
```

This operation allocates a list of the specified size, and clears it for initial use. List items may be added to the list using the **add_item** routine. Alternatively, they may be added by indexing directly into the list structure. A mixture of the two approaches for initializing a list, however, is not supported.

An **NVList** is a partially opaque structure. It may only be allocated via a call to **create_list**.

6.4.2 add_item

```
ORBStatus add_item (                                                      // PIDL
    in Identifier        item_name,          // name of item
    in TypeCode          item_type,          // item datatype
    in void              *value,             // item value
    in long              value_len,          // length of item value
    in Flags             item_flags          // item flags
);
```

This operation adds a new item to the indicated list. The item is added after the previously added item.

In addition to the argument modes defined in §6.1.1 on page 106, **item_flags** may also take the following flag values: IN_COPY_VALUE, DEPENDENT_LIST. The argument passing flags defined in §6.1.1 on page 106 may be used here to indicate the intended parameter passing mode of an argument.

If the IN_COPY_VALUE flag is set, a copy of the argument value is made and used instead.

If a list structure is added as an item (e.g. a "sublist") the DEPENDENT_LIST flag may be specified to indicate that the sublist should be freed when the parent list is freed.

6.4.3 free

```
ORBStatus free ( );                                                    // PIDL
```

This operation frees the list structure and any associated memory (an implicit call to the list **free_memory** operation is done).

6.4.4 free_memory

```
ORBStatus free_memory ( );                                             // PIDL
```

This operation frees any dynamically allocated out-arg memory associated with the list. The list structure itself is not freed.

6.4.5 get_count

```
ORBStatus get_count (                                                  // PIDL
    out long                count           // number of entries in the list
);
```

This operation returns the total number of items allocated for this list.

6.4.6 create_operation_list

This operation, which creates a pseudo-object, is defined in the ORB interface.

```
ORBStatus create_operation_list (                                      // PIDL
    in OperationDef         oper,           // operation
    out NVList              new_list        // argument definitions
);
```

This operation returns an **`NVList`** initialized with the argument descriptions for a given operation. The information is returned in a form that may be used in dynamic invocation requests. The arguments are returned in the same order as they were defined for the operation.

The list **free** operation is used to free the returned information.

6.5 Context Objects

A context object contains a list of properties, each consisting of a name and a string value associated with that name. By convention, context properties represent information about the client, environment, or circumstances of a request that are inconvenient to pass as parameters.

Context properties can represent a portion of a client's or application's environment that is meant be propagated to (and made implicitly part of) a server's environment (for example, a window identifier, or user preference information). Once a server has been invoked (i.e., subsequent to the properties being propagated), the server may query its context object for these properties.

In addition, the context associated with a particular operation is passed as a distinguished parameter, allowing particular ORBs to take advantage of context properties, for example, using the values of certain properties to influence method binding behavior, server location, or activation policy.

An operation definition may contain a clause specifying those context properties that may be of interest to a particular operation. These context properties comprise the minimum set of properties that will be propagated to the server's environment (although a specified property may have no value associated with it). The ORB may choose to pass more properties than those specified in the operation declaration.

When a context clause is present on an operation declaration, an additional argument is added to the stub and skeleton interfaces. When an operation invocation occurs via either the stub or dynamic invocation interface, the ORB causes the properties which were named in the operation definition in IDL and which are present in the client's context object, to be provided in the context object parameter to the invoked method.

Context property names (which are strings) typically have the form of an IDL identifier, or a series of IDL identifiers separated by periods. A context property name pattern is either a property name, or a property name followed by a single "*". Property name patterns are used in the **context** clause of an operation definition, and in the **get_values** operation (described below).

A property name pattern without a trailing "*" is said to match only itself. A property name pattern of the form "<name>*" matches any property name that starts with <name> and continues with zero or more additional characters.

Context objects may be created and deleted, and individual context properties may be set and retrieved. There will often be context objects associated with particular processes, users, or other things depending on the operating system, and there may be conventions for having them supplied to calls by default.

It may be possible to keep context information in persistent implementations of context objects, while other implementations may be transient. The creation and modification of persistent context objects, however, is not addressed in this specification.

Context objects may be "chained" together to achieve a particular defaulting behavior.

Properties defined in a particular context object effectively override those properties in the next higher level. This searching behavior may be restricted by specifying the appropriate scope and the "restrict scope" option on the Context **get_values** call.

Context objects may be named for purposes of specifying a starting search scope.

6.6 Context Object Routines

When performing operations on a context object, properties are represented as named value lists. Each property value corresponds to a named value item in the list.

A property name is represented by a string of characters (see §4.1.3 on page 50 for the valid set of characters that are allowed). Property names are stored preserving their case, however names cannot differ simply by their case.

The Context interface is shown below.

```
interface Context {                                                        // PIDL
    ORBStatus set_one_value (
        in Identifier       prop_name,        // property name to add
        in string           value             // property value to add
    );
    ORBStatus set_values (
        in NVList           values            // property values to be changed
    );
    ORBStatus get_values (
        in Identifier       start_scope,      // search scope
        in Flags            op_flags,         // operation flags
        in Identifier       prop_name,        // name of property(s) to retrieve
        out NVList          values            // requested property(s)
    );
    ORBStatus delete_values (
        in Identifier       prop_name         // name of property(s) to delete
    );
    ORBStatus create_child (
        in Identifier       ctx_name,         // name of context object
        out Context         child_ctx         // newly created context object
    );
    ORBStatus delete (
        in Flags            del_flags         // flags controlling deletion
    );
};
```

6.6.1 get_default_context

This operation, which creates a Context pseudo-object, is defined in the ORB interface (see §8.1 on page 143 for the complete ORB definition).

```
ORBStatus get_default_context (                                            // PIDL
    out Context             ctx               // context object
);
```

This operation returns a reference to the default process context object. The default context object may be chained into other context objects. For example, an ORB implementation may chain the default context object into its User, Group, and System context objects.

6.6.2 set_one_value

```
ORBStatus set_one_value (                                                  // PIDL
    in Identifier           prop_name,        // property name to add
    in string               value             // property value to add
);
```

This operation sets a single context object property.

At this time, only string values are supported by the context object.

6.6.3 set_values

```
ORBStatus set_values (                                                    // PIDL
    in NVList          values          // property values to be changed
);
```

This operation sets one or more property values in the context object. In the NVList, the flags field must be set to zero, and the TypeCode field associated with an attribute value must be TC_string.

At this time, only string values are supported by the context object.

6.6.4 get_values

```
ORBStatus get_values (                                                    // PIDL
    in Identifier      start_scope,    // search scope
    in Flags           op_flags,       // operation flags
    in Identifier      prop_name,      // name of property(s) to retrieve
    out NVList         values          // requested property(s)
);
```

This operation retrieves the specified context property value(s). If **prop_name** has a trailing wildcard character ("*"), then all matching properties and their values are returned. The values returned may be freed by a call to the list **free** operation.

If no properties are found an error is returned, and no property list is returned.

Scope indicates the context object level at which to initiate the search for the specified properties (e.g. "_USER", "_SYSTEM"). If the property is not found at the indicated level, the search continues up the context object tree until a match is found or all context objects in the chain have been exhausted.

Valid scope names are implementation specific.

If scope name is omitted, the search begins with the specified context object. If the specified scope name is not found, an exception is returned.

The following operation flags may be specified:

CTX_RESTRICT_SCOPE

Searching is limited to the specified search scope or context object.

6.6.5 delete_values

```
ORBStatus delete_values (                                              // PIDL
    in Identifier       prop_name        // name of property(s) to delete
);
```

This operation deletes the specified property value(s) values from the context object. If **prop_name** has a trailing wildcard character ("*"), then all property names that match will be deleted.

Search scope is always limited to the specified context object.

If no matching property is found, an exception is returned.

6.6.6 create_child

```
ORBStatus create_child (                                               // PIDL
    in Identifier       ctx_name,      // name of context object
    out Context         child_ctx      // newly created context object
);
```

This operation creates a child context object.

The returned context object is chained into its parent context. That is, searches on the child context object will look in the parent context (and so on, up the context tree), if necessary, for matching property names.

Context object names follow the rules for IDL identifiers (see §4.1.3 on page 50).

6.6.7 delete

```
ORBStatus delete (                                                     // PIDL
    in Flags            del_flags            // flags controlling deletion
);
```

This operation deletes the indicated context object.

The following option flags may be specified:

CTX_DELETE_DESCENDENTS
Deletes the indicated context object and all of its descendent context objects, as well.

An exception is returned if there are one or more child context objects and the CTX_DELETE_DESCENDENTS flag was not set.

6.7 Native Data Manipulation

A future version of this specification will define routines to facilitate the conversion of data between the list layout found in **NVList** structures and the compiler native layout.

7 The Interface Repository

The Interface Repository is the component of the ORB that provides persistent storage of interface definitions—it manages and provides access to a collection of object definitions specified in IDL.

7.1 Overview

An ORB provides distributed access to a collection of objects using the objects' publicly defined interfaces specified in IDL. The Interface Repository provides for the storage, distribution, and management of a collection of related objects' interface definitions.

For an ORB to function correctly (for it to correctly process requests), it must have access to the definitions of the objects it is handling. Object definitions can be made available to an ORB in one of two forms:

1. By incorporating the information procedurally into stub routines (e.g., as code that maps C language subroutines into communication protocols).
2. As objects accessed through the dynamically accessible Interface Repository (i.e., as "interface objects" accessed through IDL-specified interfaces).

In particular, the ORB can use object definitions maintained in the Interface Repository to interpret/handle the values provided in a request:

- To provide type-checking of request signatures (whether the request was issued through the API or through a stub).
- To assist in checking the correctness of interface inheritance graphs.
- To assist in providing interoperability between different ORB implementations.

As the interface to the object definitions maintained in an Interface Repository is public, the information maintained in the Repository can also be used by clients and services. For example, the Repository can be used:

- To manage the installation and distribution of interface definitions.
- To provide components of a CASE environment (e.g., an interface browser).
- To provide interface information to language-bindings (e.g., a compiler).
- To provide components of enduser environments (e.g., a menu bar constructor).

7.2 Scope of an Interface Repository

Interface definitions are maintained in the Interface Repository as a set of objects that are accessible through a set of IDL-specified interface definitions. An interface definition contains a description of the operations it supports, including the types of the parameters, exceptions it may raise, and context information it may use.

In addition, the interface repository stores constant values, which might be used in other interface definitions or might simply be defined for programmer convenience. And it stores typecodes, which are values that describe a type in structural terms.

The Interface Repository uses modules as a way to group interfaces and to navigate through those groups by name. Modules can contain constants, typedefs, exceptions, interface definitions, and other modules. The standard does not specify what constitutes the group of interfaces that comprises a particular module. Modules may, for example, correspond to the organization of IDL definitions. They may also be used to represent organizations defined for administration or other purposes.

The Interface Repository is a set of objects that represent the information in it. There are operations that operate on this apparent object structure. It is an implementation's choice whether these objects exist persistently or are created when referenced in an operation on the repository. There are also operations that extract information in an efficient form, obtaining a block of information that describes a whole interface or a whole operation.

An ORB may have access to multiple Interface Repositories. This may occur because two ORBs have different requirements for the implementation of the Interface Repository, because an object implementation (such as an OODB) prefers to provide its own type information, or because it is desired to have different additional information stored in different repositories. The use of typecodes and repository identifiers is intended to allow different repositories to keep their information consistent.

This CORBA Interface Repository specification defines operations only for retrieving information from the repository. There may be any number of ways to insert information into the repository (for example, compiling IDL definitions, constructing repository objects through the dynamic invocation interface, copying objects from one repository to another, etc.). Because some of these scenarios involve development environments, system administration, and other issues beyond the scope of the ORB, this part of the interface repository is left undefined.

7.3 Implementation Dependencies

An implementation of an Interface Repository requires some form of persistent object store. Normally the kind of persistent object store used determines how interface definitions are distributed and/or replicated throughout a network domain. For example, if an Interface Repository is implemented using a filing system to provide object storage, there may be only a single copy of a set of interfaces maintained on a single machine. Alternatively, if an OODB is used to provide object storage, multiple copies of interface definitions may be maintained each of which is distributed across several machines to provide both high-availability and load-balancing.

The kind of object store used may determine the scope of interface definitions provided by an implementation of the Interface Repository. For example, it may determine whether each user has a local copy of a set of interfaces or if there is one copy per community of users. The object store may also determine whether or not all clients of an interface set see exactly the same set at any given point in time or whether latency in distributing copies of the set gives different users different views of the set at any point in time.

An implementation of the Interface Repository is also dependent on the security mechanism in use. The security mechanism (usually operating in conjunction with the object store) determines the nature and granularity of access controls available to constrain access to objects in the repository.

7.4 Basics of the Interface Repository Interface

7.4.1 Names

Names are not necessarily unique within an Interface Repository; they are always relative to an explicit or implicit module. In this context, interface definitions are considered explicit modules.

Repository identifiers uniquely identify modules, interfaces, constants, typedefs, exceptions, attributes, operations and parameters within a particular Interface Repository.

7.4.2 Types and TypeCodes

The Interface Repository stores information about types that are not interfaces in a data value called a TypeCode. From the TypeCode alone it is possible to determine the complete structure of a type. See §7.6 on page 138 for more information on the internal structure of TypeCodes.

7.4.3 Interface Objects

Each interface managed in an Interface Repository is maintained as a collection of interface objects:

1. Repository: the top-level module for the repository name space; it contains constants, typedefs, exceptions, interface definitions, and modules.
2. ModuleDef: a logical grouping of interfaces; it contains constants, typedefs, exceptions, interface definitions, and other modules.
3. InterfaceDef: an interface definition; it contains lists of constants, types, exceptions, operations, and attributes.
4. AttributeDef: the definition of an attribute of the interface.
5. OperationDef: the definition of an operation on the interface; it contains lists of parameters and exceptions raised by this operation.
6. ParameterDef: the definition of an argument to an operation.
7. TypeDef: the definition of named type that is not an interface.
8. ConstantDef: the definition of a named constant.
9. ExceptionDef: the definition of an exception that can be raised by an operation.

The interface specifications for each interface object lists the attributes maintained by that object (see §7.5 on page 129). These attributes correspond directly to IDL statements. An implementation can choose to maintain additional attributes to facilitate managing the Repository or to record additional (proprietary) information about an interface. Note that in practice, attributes used to define an Interface Repository object will be read-only attributes.

The interface specifications for the Interface Repository objects define a set of basic operations for clients who want to access these interface objects. They are not intended to provide sufficient semantics for the construction of basic interface browsers or command-line interfaces to the Interface Repository, nor to provide an administrative interface. Implementations of the Interface Repository must have additional operations for creation of the component objects within an Interface object, but these need not be part of the public interface.

The CORBA specification defines a minimal set of operations for interface objects. Additional operations that an implementation of the Interface Repository may provide could include operations that provide for the versioning of interfaces, for the deletion of interfaces, and for the reverse compilation of specifications (i.e., the generation of a file containing an object's IDL specification).

7.4.4 Structure and Navigation of Interface Objects

The definitions in the Interface Repository are structured as a set of objects. The objects are structured the same way definitions are structured—some objects (definitions) "contain" other objects.

The containment relationships for the objects in the Interface Repository are shown in FIG. 12 on page 128.

FIG. 12 Interface Repository Object Containment

Repository	**Each interface repository is represented by a global root repository object.**
ConstantDef **TypeDef** **ExceptionDef** **InterfaceDef** **ModuleDef**	**The repository object represents the constants, typedefs, exceptions, interfaces and modules that are defined outside the scope of a module.**
ConstantDef **TypeDef** **ExceptionDef** **ModuleDef** **InterfaceDef**	**The module object represents the constants, typedefs, exceptions, interfaces, and other modules defined within the scope of the module.**
ConstantDef **TypeDef** **ExceptionDef** **AttributeDef** **OperationDef**	**An interface object represents constants, typedefs, exceptions, attributes, and operations defined within or inherited by the interface.** **Operation objects reference parameter and exception objects.**

There are three ways to locate an interface in the Interface Repository:

1. By obtaining an InterfaceDef object directly from the ORB (see §8.2 on page 145).
2. By navigating through the module name space using a sequence of names.
3. By locating the InterfaceDef object that corresponds to a particular repository identifier.

Obtaining an InterfaceDef object directly is useful when an object is encountered whose type was not known at compile time. By using the **get_interface()** operation on the object reference, it is possible to retrieve the Interface Repository information about the object. That information could then be used to perform operations on the object.

Navigating the module name space is useful when information about a particular named interface is desired. Starting at the root module of the repository, it is possible to obtain entries by name.

Locating the InterfaceDef object by ID is useful when looking for an entry in one repository that corresponds to another. A repository identifier must be unique for a particular repository. By using the same identifier in two repositories, it is possible to obtain the interface identifier for an interface in one repository, and then obtain information about that interface from another repository that may be closer or contain additional information about the interface.

7.5 Interface Repository Interfaces

A common set of operations is used to locate objects within the Interface Repository. These operations are defined in the abstract interfaces **Container** and **Contained** described below. Objects that are containers inherit navigation operations from the **Container** interface. Objects that are contained by other objects inherit navigation operations from the **Contained** interface. The **Contained** and **Container** interfaces are not instantiable.

7.5.1 Container and Contained Interfaces

The following typedefs are used in the definitions of the **Contained** and **Container** interfaces:

```
typedef string          Identifier;
typedef string          RepositoryId;
typedef Identifier      InterfaceName;
```

Identifiers are the simple names that identify modules, interface, constants, typedefs, exceptions, attributes, and operations. They correspond exactly to the identifiers described in §4.1.3 on page 50 and §4.3 on page 54. For example, both "**SampleDef**" and "**SampleOp**" are **identifier**s in the following IDL definition.

```
interface SampleDef {
    void SampleOp ();
};
```

A name is not necessarily unique within an Interface Repository. As described in §4.11 on page 76, names are unique only within a module.

A **RepositoryId** is an identifier used by the ORB to uniquely identify module, interface, constant, typedef, exception, attribute or operation. While **RepositoryId**s correspond to global or fully qualified names (with respect to a particular repository), they are opaque structures so that an ORB may optimize their implementation. As **RepositoryId**s are defined as strings, they can be manipulated (e.g., copied and compared) using a language binding's string manipulation routines.

An **InterfaceName** is a string containing the name of one of the following interfaces defined within the Interface Repository:

"AttributeDef", "ConstantDef", "ExceptionDef", "InterfaceDef", "ModuleDef", "ParameterDef", "OperationDef", TypeDef", "all"

The **Contained** interface represents the most generic form of interface from which all other Interface Repository interfaces are derived. All objects within the Interface Repository, except the root object (the Interface Repository itself) can be contained by other objects.

```
interface Contained {
    attribute Identifier        name;
    attribute RepositoryId      id;
    attribute RepositoryId      defined_in;

    sequence <Container> within ();

    struct Description {
        Identifier    name;
        any           value;
    };
    Description describe ();
};
```

All objects that are contained by other objects have a **name** and an **id** to identify them.

Objects can be contained either because they are defined within the containing object (for example, an interface is defined within a module) or because they are inherited by the containing object (for example, an operation may be contained by an interface because the interface inherits the operation from another interface). If an object is contained through inheritance, the **defined_in** attribute identifies the interface of the object from which the object is derived. If the object is not inherited (it is defined here), this attribute contains the **id** of this interface or module.

The **within** operation returns the list of objects that contain the object. If the object is an interface or module it can be contained only by the object that defines it. Other objects can be contained by the objects that define them and by the objects that inherit them.

The **describe** operation returns a structure containing all of the attributes defined for the interface. The **describe** structure associated with each interface is provided below with the interface's definition. The name of the structure returned is provided with the returned structure. For example, if the **describe** operation is invoked on an attribute object, the **name** field contains "AttributeDescription" and the **value** field contains an **any**, which contains the AttributeDescription structure.

The **Container** interface is used to locate objects that are contained by other objects.

```
interface Container {
    sequence<Contained> contents (
        in InterfaceName        limit_type,
        in boolean              exclude_inherited
    );
    sequence<Contained> lookup_name (
        in Identifier           search_name,
        in long                 levels_to_search,
        in InterfaceName        limit_type,
        in boolean              exclude_inherited
    );
    struct Description {
        Contained               contained_object;
        Identifier              name;
        any                     value;
    };
    sequence<Description> describe_contents(
        in InterfaceName        limit_type,
        in boolean              exclude_inherited,
        in long                 max_returned_objs
    );
};
```

The **contents** operation returns the list of objects contained by the object. The operation is used to navigate through the hierarchy of objects. Starting with the Repository object, a client uses this operation to list all of the objects contained by the Repository, all of the objects contained by the modules within the Repository, and then all of the interfaces within a specific module, and so on.

restrict_type
: If **restrict_type** is set to “all”, objects of all interface types are returned. For example, if this is an InterfaceDef, the attribute, operation, and exception objects are all returned. If **restrict_type** is set to a specific interface, only objects of that interface type are returned. For example, only attribute objects are returned if **restrict_type** is set to “AttributeDef”.

exclude_inherited
: If set to TRUE, inherited objects (if there are any) are not returned. If set to FALSE, all contained objects—whether contained due to inheritance or because they were defined within the object—are returned.

The **lookup_name** operation is used to locate an object by name within a particular object or within the objects contained by that object.

search_name Specifies which name is to be searched for.

levels_to_search Controls whether the lookup is constrained to the object the operation is invoked on or whether it should search through objects contained by the object as well.

Setting **levels_to_search** to -1 searches the current object and all contained objects. Setting **levels_to_search** to 1 searches only the current object. Setting **levels_to_search** to any other positive integer constrains the lookup to only search through containers that many levels down.

restrict_type If **restrict_type** is set to "all", objects of all interface types are returned (e.g., attributes, operations, and exceptions are all returned). If **restrict_type** is set to a specific interface, only objects of that interface type are returned. For example, only attribute objects are returned if **restrict_type** is set to "AttributeDef".

exclude_inherited If set to TRUE, inherited objects (if there are any) are not returned. If set to FALSE, all contained objects (whether contained due to inheritance or because they were defined within the object) are returned.

The **describe_contents** operation combines the **contents** operation and the **describe** operation. For each object returned by the **contents** operation, the description of the object is returned (i.e., the object's **describe** operation is invoked and the results returned).

max_returned_objs Limits the number of objects that can be returned in an invocation of the call to the number provided. Setting the parameter to -1 means return all contained objects.

7.5.2 Repository

Repository is an interface that provides global access to the Interface Repository. The **Repository** object contains constants, typedefs, exceptions, interfaces, and modules. As it inherits from **Container**, it can be used to look up any definition (whether globally defined or defined within a module or interface) either by **name** or by **id**.

There may be more than one Interface Repository in a particular ORB environment (although some ORBs might require that definitions they use be registered with a particular repository). Each ORB environment will provide a means for obtaining object references to the Repositories available within the environment.

```
interface Repository : Container {
    Contained lookup_id (in RepositoryId search_id);
};
```

The **lookup_id** operation is used to lookup an object in the Repository given its **RepositoryId**.

The inherited **contents** operation can be used to list constants, typedefs, exceptions, interfaces, and modules that are contained by the Repository object, or any one of these interface types. Constants, typedefs, exceptions, and interfaces contained by the Repository object have global scope throughout the Interface Repository.

The inherited **lookup_name** operation can be used to find instances of a particular simple name within an Interface Repository. For example, specifying a **search_name** of “Print” and a **levels_to_search** of -1 would return a list of all the Print operations defined within this Interface Repository.

The inherited **describe_contents** operation can be used to list a more complete description of the contents of the Interface Repository. For each object, **describe_contents** returns:

```
struct RepositoryDescription {
    Identifier      name;
    RepositoryId    id;
    RepositoryId    defined_in;
};
```

7.5.3 ModuleDef

A ModuleDef represents constants, typedefs, exceptions, interfaces and other module objects.

```
interface ModuleDef : Container, Contained {};
```

The **describe** operation for a ModuleDef object returns:

```
struct ModuleDescription {
    Identifier      name;
    RepositoryId    id;
    RepositoryId    defined_in;
};
```

The **lookup_name** operation is used to locate a constant, typedef, exception, attribute, or operation defined in this module. Names must be unique within a module.

7.5.4 InterfaceDef

An InterfaceDef object represents an interface definition.

```
interface InterfaceDef : Container, Contained {
    sequence<RepositoryId>    base_interfaces;

    struct FullInterfaceDescription {
        Identifier                          name;
        RepositoryId                        id;
        RepositoryId                        defined_in;
        sequence<OperationDescription>      operation
        sequence<AttributeDescription>      attribute;
    };

    FullInterfaceDescription describe_interface();
};
```

The **base_interfaces** attribute represents a list of all the interfaces from which this interface inherits.

The **describe** operation for an InterfaceDef object returns:

```
struct InterfaceDescription {
    Identifier        name;
    RepositoryId      id;
    RepositoryId      defined_in;
};
```

The **contents** operation returns the list of constants, typedefs, and exceptions defined in this InterfaceDef and the list of attributes and operations either defined or inherited in this InterfaceDef. If the **exclude_inherited** parameter is set to TRUE, only attributes and operations defined within this interface are returned. If the **exclude_inherited** parameter is set to FALSE, all attributes and operations are returned.

The **lookup_name** operation is used to locate a constant, typedef, exception, attribute, operation, or parameter defined in this InterfaceDef by name. Names must be unique within a interface definition.

The **describe_contents** operation provides a complete description of this interface, including a description of its constants, typedefs, exceptions, attributes, and operations, including the description of each parameter and each exception defined for each operation.

The **describe_interface** operation provides a convenient way to obtain a description of all of the operations and attributes defined in an interface.

7.5.5 AttributeDef

An AttributeDef represents the information that defines an attribute.

```
interface AttributeDef : Contained {
    enum AttributeMode {NORMAL, READONLY};

    attribute TypeCode          type;
    attribute AttributeMode     mode;
};
```

The **type** attribute specifies the TypeCode of this attribute. The TypeCode must be known globally, within this module, or defined in this interface. See §7.6 on page 138 for more information on TypeCodes.

The **mode** attribute specifies read only or read/write access for this attribute.

The **describe** operation for an AttributeDef object returns:

```
struct AttributeDescription {
    Identifier          name;
    RepositoryId        id;
    RepositoryId        defined_in;
    TypeCode            type
    AttributeMode       mode;
};
```

7.5.6 OperationDef

An OperationDef represents the information needed to define an operation.

```
interface OperationDef : Contained {
    typedef Identifier          ContextIdentifier;
    enum OperationMode          {NORMAL, ONEWAY};

    attribute TypeCode                              result;
    attribute OperationMode                         mode;
    attribute sequence<ContextIdentifier>           context;
};
```

The **result** attribute specifies the TypeCode of the value returned by this operation.

The operation’s **mode** is either “oneway” (i.e., no output is returned) or normal.

The **context** attribute specifies the list of context identifiers that apply to this operation.

The **describe** operation for an OperationDef object returns:

```
struct OperationDescription {
    Identifier                        name;
    RepositoryId                      id;
    RepositoryId                      defined_in;
    TypeCode                          result;
    OperationMode                     mode;
    sequence<string>                  context;
    sequence<ParameterDescription>    parameter;
    sequence<ExceptionDescription>    exception
};
```

The **describe_contents** operation provides a complete description of this operation, including a description of each parameter and each exception defined for this operation.

7.5.7 ParameterDef

A ParameterDef represents the information needed to define an argument to an operation.

```
interface ParameterDef : Contained {
    enum ParameterMode {IN, OUT, INOUT};

    attribute TypeCode           type;
    attribute ParameterMode      mode;
};
```

The **type** attribute specifies the TypeCode of this argument. The TypeCode must be known globally, within this module, or defined in this interface. See §7.6 on page 138 for more information on TypeCodes.

The **mode** attribute specifies whether this argument is used as an in, out, or in/out argument.

The **describe** operation for a ParameterDef object returns:

```
struct ParameterDescription {
    Identifier          name;
    RepositoryId        id;
    RepositoryId        defined_in;
    TypeCode            type;
    ParameterMode       mode;
};
```

7.5.8 TypeDef Interface

TypeDefs are only for named types (i.e., TypeDef, Struct, ...) They aren't created for in-line types.

```
interface TypeDef : Contained {
    attribute TypeCode      type;
};
```

The **type** attribute specifies the TypeCode for this type definition. See §7.6 on page 138 for more information on TypeCodes.

The **describe** operation for a TypeDef object returns:

```
struct TypeDescription {
    Identifier              name;
    RepositoryId            id;
    RepositoryId            defined_in;
    TypeCode                type;
};
```

7.5.9 ConstantDef Interface

The ConstantDef object defines a named constant.

```
interface ConstantDef : Contained {
    attribute TypeCode      type;
    attribute any           value;
};
```

The **type** attribute specifies the TypeCode for this constant. The type of a constant must be one of the simple types (long, short, float, char, string, octet, etc.)

The **value** attribute contains the value of the constant, not the computation of the value (e.g., the fact that it was defined as "1+2").

The **describe** operation for a ConstantDef object returns:

```
struct ConstantDescription {
    Identifier              name;
    RepositoryId            id;
    RepositoryId            defined_in;
    TypeCode                type;
    any                     value;
};
```

7.5.10 ExceptionDef

An ExceptionDef represents an exception definition.

```
interface ExceptionDef : Contained {
    attribute TypeCode          type;
};
```

The **type** attribute specifies the type of the exception parameters. The type will be a struct with the members corresponding to the individual exception parameters.

The **describe** operation for a ExceptionDef object returns:

```
struct ExceptionDescription {
    Identifier          name;
    RepositoryId        id;
    RepositoryId        defined_in;
    TypeCode            type;
};
```

7.6 TypeCodes

TypeCodes are values that represent invocation argument types and attribute types. They can be obtained from the Interface Repository or from IDL compilers.

TypeCodes have a number of uses. They are used in the dynamic invocation interface to indicate the types of the actual arguments. They are used by an Interface Repository to represent the type specifications that are part of many IDL declarations. Finally, they are crucial to the semantics of the **any** type (see §4.7.1.6 on page 68).

TypeCodes are themselves values that can be passed as invocation arguments. To allow different ORB implementations to hide extra information in TypeCodes, the representation of TypeCodes will be opaque (like object references). However, we will assume that the representation is such that TypeCode “literals” can be placed in C include files.

Abstractly, TypeCodes consist of a “kind” field, and a “parameter list.” For example, the IDL type **long** has TypeCode tk_long and no parameters. The IDL type **sequence<boolean,10>** has TypeCode tk_sequence and two parameters: **10** and **boolean**.

Two TypeCodes are equal if the IDL type specifications from which they are compiled denote equal types. Equal TypeCodes are interchangeable, and give identical results when TypeCode operations are applied to them.

7.6.1 The TypeCode Interface

The IDL interface for TypeCodes is

```
interface TypeCode {
    enum TCKind {
        tk_null, tk_void,
        tk_short, tk_long, tk_ushort, tk_ulong,
        tk_float, tk_double, tk_boolean, tk_char,
        tk_octet, tk_any, tk_TypeCode, tk_Principal, tk_objref,
        tk_struct, tk_union, tk_enum, tk_string,
        tk_sequence, tk_array
    };

    exception Bounds {};
    boolean equal (in TypeCode tc);
    TCkind kind ();
    long param_count ();
        // The number of parameters for this TypeCode.
    any parameter (in long index) raises (Bounds);
        // The index'th parameter. Parameters are indexed from 0
        // to (param_count-1).
}
```

With the above operations, any TypeCode can be decomposed into its constituent parts. Composing new TypeCodes is a topic for future OMG standardization.

The legal combinations of kinds and parameters are listed in TBL. 16 on page 139.

TBL. 16 Legal TypeCode Kinds and Parameters

KIND	PARAMETER LIST
tk_null	*NONE*
tk_void	*NONE*
tk_short	*NONE*
tk_long	*NONE*
tk_ushort	*NONE*
tk_ulong	*NONE*
tk_float	*NONE*
tk_double	*NONE*
tk_boolean	*NONE*
tk_char	*NONE*
tk_octet	*NONE*
tk_any	*NONE*

TBL. 16 Legal TypeCode Kinds and Parameters (Continued)

KIND	PARAMETER LIST
tk_TypeCode	*NONE*
tk_Principal	*NONE*
tk_objref	{ interface-id }
tk_struct	{struct-name, member-name, TypeCode, ... (repeat pairs) }
tk_union	{union-name, switch-TypeCode, label-value, member-name, TypeCode, ... (repeat triples) }
tk_enum	{enum-name, enum-id, ... }
tk_string	{ maxlen-integer }
tk_sequence	{ TypeCode, maxlen-integer }
tk_array	{ TypeCode, length-integer }

The tk_objref TypeCode represents an interface type. Its parameter is the interface-id of that interface.

A structure with N members results in a tk_struct TypeCode with 2N+1 parameters: the type name of the struct, the rest are member names alternating with the corresponding member TypeCode. Member names are represented as strings.

A union with N members results in a tk_union TypeCode with 3N+2 parameters: the type name of the union, the switch TypeCode followed by a label value, member name, and member TypeCode for each of the N members. The label values are all values of the data type designated by the switch TypeCode, with one exception. The default member (if present) is marked with a label value consisting of the 0 **octet**. Recall that the operation "parameter(tc,i)" returns an **any**, and that anys themselves carry a TypeCode that can distinguish an octet from any of the legal switch types.

The tk_enum TypeCode has the type name of the enum followed by the enumeration ids as parameters. Enumeration ids are represented as strings.

The tk_string TypeCode has 1 parameter: an integer giving the maximum string length. A maximum of 0 denotes unbounded.

The tk_sequence TypeCode has 2 parameters: a TypeCode for the sequence elements, and an integer giving the maximum sequence. Again, 0 denotes unbounded.

Finally, the tk_array TypeCode has 2 parameters: a TypeCode for the array elements, and an integer giving the array length. Arrays are never unbounded.

7.6.2 TypeCode Constants

If "**typedef ... FOO;**" is an IDL type declaration, the IDL compiler will (if asked) produce a declaration of a TypeCode constant named TC_FOO. This constant can be used with the dynamic invocation interface, and any other routines that require TypeCodes. The predefined TypeCode constants are:

TC_null		
TC_void		
TC_short		
TC_long		
TC_ushort		
TC_ulong		
TC_float		
TC_double		
TC_boolean		
TC_char		
TC_octet		
TC_any		
TC_TypeCode		
TC_Principal		
TC_Object	= tk_objref { Object }	
TC_string	= tk_string { 0 }	// unbounded
TC_NamedValue	= tk_struct { ... }	
TC_InterfaceDescription	= tk_struct { ... }	
TC_OperationDescription	= tk_struct { ... }	
TC_AttributeDescription	= tk_struct { ... }	
TC_ParameterDescription	= tk_struct { ... }	
TC_RepositoryDescription	= tk_struct { ... }	
TC_ModuleDescription	= tk_struct { ... }	
TC_ConstDescription	= tk_struct { ... }	
TC_ExceptionDescription	= tk_struct { ... }	
TC_TypeDescription	= tk_struct { ... }	
TC_FullInterfaceDescription	= tk_struct { ... }	

The exact form for TypeCode constants is language mapping, and possibly implementation, specific.

8 ORB Interface

The ORB interface is the interface to those ORB functions that do not depend on which object adapter is used. These operations are the same for all ORBs and all object implementations, and can be performed either by clients of the objects or implementations. Some of these operations appear to be on the ORB, others appear to be on the object reference. Because the operations in this section are implemented by the ORB itself, they are not in fact operations on objects, although they may be described that way and the language binding will, for consistency, make them appear that way.

The ORB interface also defines operations for creating lists and determining the default context used in the Dynamic Invocation Interface. Those operations are described in Chapter 6.

8.1 Converting Object References to Strings

Because an object reference is opaque and may differ from ORB to ORB, the object reference itself is not a convenient value for storing references to objects in persistent storage or communicating references by means other than invocation. Two problems must be solved: allowing an object reference to be turned into a value that a client can store in some other medium, and ensuring that the value can subsequently be turned into the appropriate object reference.

An object reference may be translated into a string by the operation **object_to_string**. The value may be stored or communicated in whatever ways strings may be manipulated. Subsequently, the **string_to_object** operation will accept a string produced by **object_to_string** and return the corresponding object reference.

```
interface ORB {                                                    // PIDL
    string object_to_string (in Object obj);
    Object string_to_object (in string str);

    ORBStatus create_list (
        in long             count,
        out NVList          new_list
    );
    ORBStatus create_operation_list (
        in OperationDef     oper,
        out NVList          new_list
    );

    ORBStatus get_default_context (out Context ctx);
};
```

To guarantee that an ORB will understand the string form of an object reference, that ORB's **object_to_string** operation must be used to produce the string. Since in general a client does not know or care which ORB is used for a particular object reference, the client can choose whatever ORB is convenient.

For a description of the **create_list** and **create_operation_list** operations, see §6.4 on page 113. The **get_default_context** operation is described in §6.6.1 on page 118.

8.2 Object Reference Operations

There are some operations that can be done on any object. These are not operations in the normal sense, in that they are implemented directly by the ORB, not passed on to the object implementation. We will describe these as being operations on the object reference, although the interfaces actually depend on the language binding. As above, where we used interface Object to represent the object reference, we will define an interface for Object:

```
interface Object {                                                    // PIDL
    ImplementationDef get_implementation ();
    InterfaceDef get_interface ();
    boolean is_nil();
    Object duplicate ();
    void release ();

    ORBStatus create_request (
        in Context          ctx,
        in Identifier       operation,
        in NVList           arg_list,
        inout NamedValue    result,
        out Request         request,
        in Flags            req_flags
    );
};
```

The **create_request** operation is part of the Object interface because it creates a pseudo-object (a Request) for an object. It is described with the other Request operations in §6.2 on page 108.

8.2.1 Determining the Object Implementation and Interface

An operation on the object reference, **get_interface**, returns an object in the Interface Repository, which provides type information that may be useful to a program. See Chapter 7 for a definition of operations on the Interface Repository. An operation on the Object called **get_implementation** will return an object in an implementation repository that describes the implementation of the object. See Chapter 9 for information about the Implementation Repository.

```
InterfaceDef get_interface ();                                        // PIDL
ImplementationDef get_implementation ();
```

8.2.2 Duplicating and Releasing Copies of Object References

Because object references are opaque and ORB-dependent, it is not possible for clients or implementations to allocate storage for them. Therefore, there are operations defined to copy or release an object reference.

```
Object duplicate ();                                                    // PIDL
void release ();
```

If more than one copy of an object reference is needed, the client may create a **duplicate**. Note that the object implementation is not involved in creating the duplicate, and that the implementation cannot distinguish whether the original or a duplicate was used in a particular request.

When an object reference is no longer needed by a program, its storage may be reclaimed by use of the **release** operation. Note that the object implementation is not involved, and that neither the object itself nor any other references to it are affected by the **release** operation.

8.2.3 Nil Object References

An object reference whose value is OBJECT_NIL denotes no object. An object reference can be tested for this value by the **is_nil** operation. The object implementation is not involved in the nil test.

```
boolean is_nil ();                                                      // PIDL
```

9 The Basic Object Adapter

An Object Adapter is the primary interface that an implementation uses to access ORB functions. The *Basic Object Adapter* (BOA) is an interface intended to be widely available and to support a wide variety of common object implementations. It includes convenient interfaces for generating object references, registering implementations that consist of one or more programs, activating implementations, and authenticating requests. It also provides a limited amount of persistent storage for objects that can be used for connecting to a larger or more general storage facility, for storing access control information, or other purposes.

Most of the Basic Object Adapter interface can be expressed in IDL, since the interface is to the operations on the object adapter. Some of the operations to bind the implementation to the object adapter depend on the language mapping. We will note such dependencies, but still use IDL as the means to describe the interface.

9.1 Role of the Basic Object Adapter

One object adapter, called the Basic Object Adapter, should be available in every ORB implementation; although the BOA will generally have an ORB-dependent implementation, object implementations that use it should be able to run on any ORB that supports the required language mapping, assuming they have been installed appropriately.

Other Object Adapters are likely to be created. Ordinarily, it is not necessary for a client of an object to be concerned about which Object Adapter is used by the implementation.

The following functions are provided through the Basic Object Adapter:

- Generation and interpretation of object references,
- Authentication of the principal making the call,
- Activation and deactivation of the implementation,
- Activation and deactivation of individual objects, and
- Method invocation through skeletons.

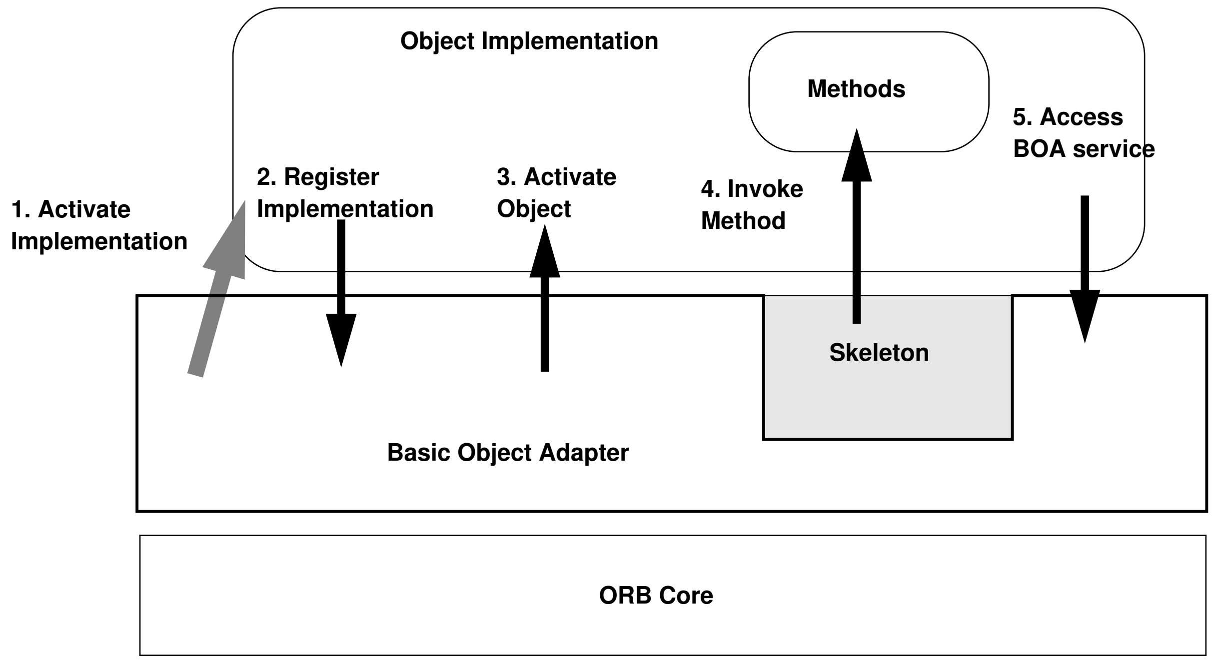
The Basic Object Adapter supports object implementations that are constructed from one or more programs[1]. The BOA activates and communicates with these programs using operating system facilities that are not part of the ORB. Therefore the BOA requires some information that is inherently non-portable. Although not defining this information, the BOA does define the concept of an Implementation Repository which can hold this information, allowing each system to install and start implementations in the way that is appropriate for that system.

The mechanism for binding the program to the BOA and ORB is also not specified because it is inherently system and language-dependent. We assume that the BOA can connect the methods to the skeleton by some means, whether at the time the implementation is compiled, installed, or activated, etc. Subsequent to activation, the BOA can make calls on routines in the implementation and the implementation can make calls on the BOA.

FIG. 13 on page 149 shows the structure of the Basic Object Adapter, and some of the interactions between the BOA and an Object Implementation. The Basic Object Adapter will start a program to provide the Object Implementation, in this example, a per-class server (1). The Object Implementation notifies the BOA that it has finished initializing and is prepared to handle requests (2). When the first request for a particular object arrives, the implementation is notified to activate the object (3). On subsequent requests, the BOA calls the appropriate method using the per-interface skeleton (4). At various times, the implementation may access BOA services such as object creation, deactivation, etc. (5).

1. The term "program" is meant to include a wide range of possible constructs, including scripts, loadable modules, etc., in addition to the traditional notions of an application or server.

FIG. 13 The Structure and Operation of the Basic Object Adapter



The BOA exports operations that are accessed by the Object Implementation. The BOA also calls the Object Implementation under certain circumstances. The interface between a particular version of the BOA and the ORB Core it runs on is private, as is the interface between the BOA and the skeletons. Thus, the BOA can exploit features or overcome limitations of a specific ORB Core, and can cooperate with the ORB Core and skeletons to provide a set of portable interfaces for the object implementation.

9.2 Basic Object Adapter Interface

The BOA interface is specified in IDL, so that the way it is accessed in any programming language is specified by the client side language mapping for that language. Some data structures used by the BOA are specific to a given language mapping, so most IDL compilers will not be able to accept this definition literally.

In practice, the BOA is most likely to be implemented partially as a separate component and partially as a library in the Object Implementation. The separate component is required to do activation when the implementation is not present. The library portion is needed to establish the linkage between the methods and the skeleton. The exact partitioning of functionality between these parts is implementation dependent. Generally, there will appear to be a BOA object in the object implementation. When it is invoked, some operations are satisfied in the library, some in an external server, and some in the ORB Core.

The following is the approximate interface definition for the BOA object. More details will be provided as the operations are discussed.

```
interface InterfaceDef;              // from Interface Repository          // PIDL
interface ImplementationDef;         // from Implementation Repository
interface Object;                    // an object reference
interface Principal;                 // for the authentication service
typedef sequence <octet, 1024> ReferenceData;

interface BOA {
    Object create (
        in ReferenceData          id,
        in InterfaceDef           intf,
        in ImplementationDef      impl
    );
    void dispose (in Object obj);
    ReferenceData get_id (in Object obj);

    void change_implementation (
        in Object                 obj,
        in ImplementationDef      impl
    );

    Principal get_principal (
        in Object                 obj,
        in Environment            ev
    );

    void set_exception (
        in exception_type         major,    // NO, USER, or SYSTEM_EXCEPTION
        in string                 userid,   // exception type id
        in void                   *param    // pointer to associated data
    );

    void impl_is_ready (in ImplementationDef impl);
    void deactivate_impl (in ImplementationDef impl);
    void obj_is_ready (in Object obj, in ImplementationDef impl);
    void deactivate_obj (in Object obj);
};
```

Requests by an implementation on the BOA are of three kinds:

1. Operations to create or destroy object references, or query or update the information the BOA maintains for an object reference.
2. Operations associated with a particular request.

3. Operations to maintain a registry of active objects and implementations.

Requests by the BOA to an implementation are made with skeletons or using an implementation's runtime language mapping information, and are of three kinds:

1. Activating an implementation.
2. Activating an object.
3. Performing an operation (through a skeleton method).

Each of the BOA operations is described in detail later in this section; the requests of the BOA to an implementation are described in the language mapping section.

9.2.1 Registration of Implementations

The Basic Object Adapter expects information describing the implementations to be stored in an *Implementation Repository.* The Implementation Repository ordinarily is updated at program installation time, but may be set up incrementally or otherwise. There are objects with an IDL interface called **ImplementationDef**, which capture this information. The Implementation Repository may contain additional information for debugging, administration, etc. Note that the Implementation Repository is logically distinct from the Interface Repository, although they may in fact be implemented together.

The *Interface Repository* contains information about interfaces. There are objects with an IDL interface called **InterfaceDef**, which capture this information. The Interface Repository may contain additional information for debugging, administration, browsing, etc. The ORB Core may or may not make use of the Interface Repository or the Implementation Repository, but the ORB and BOA use these objects to associate object references with their interfaces and implementations.

9.2.2 Activation and Deactivation of Implementations

There are two kinds of activation that a BOA needs to perform as part of operation invocation. The first, discussed in this section, is *implementation activation*, which occurs when no implementation for an object is currently available to handle the request. The second, discussed later, is *object activation*, which occurs when no instance of the object is available to handle the request.

Implementation activation requires coordination between the BOA and the program(s) containing the implementation. We use the term *server* as the separately executable entity that the BOA can start on a particular system. In a POSIX environment, a server would be a process. In most systems, a server corresponds to the notion of a program, but it can correspond to whatever the appropriate system facility is in a particular environment.

The BOA initiates activity by the implementation by starting the appropriate server, probably in an operating system-dependent way. The implementation initializes itself, then notifies the BOA that it is prepared to handle requests by calling **impl_is_ready** or **obj_is_ready**[2]. Between the time that the program is started and it indicates it is ready, the BOA will prevent any other requests from being delivered to the server. After that point, the BOA, through the skeletons, will make calls on the methods of the implementation.

```
void impl_is_ready (in ImplementationDef impl);                    // PIDL
void obj_is_ready (
    in Object               obj,
    in ImplementationDef    impl
);
```

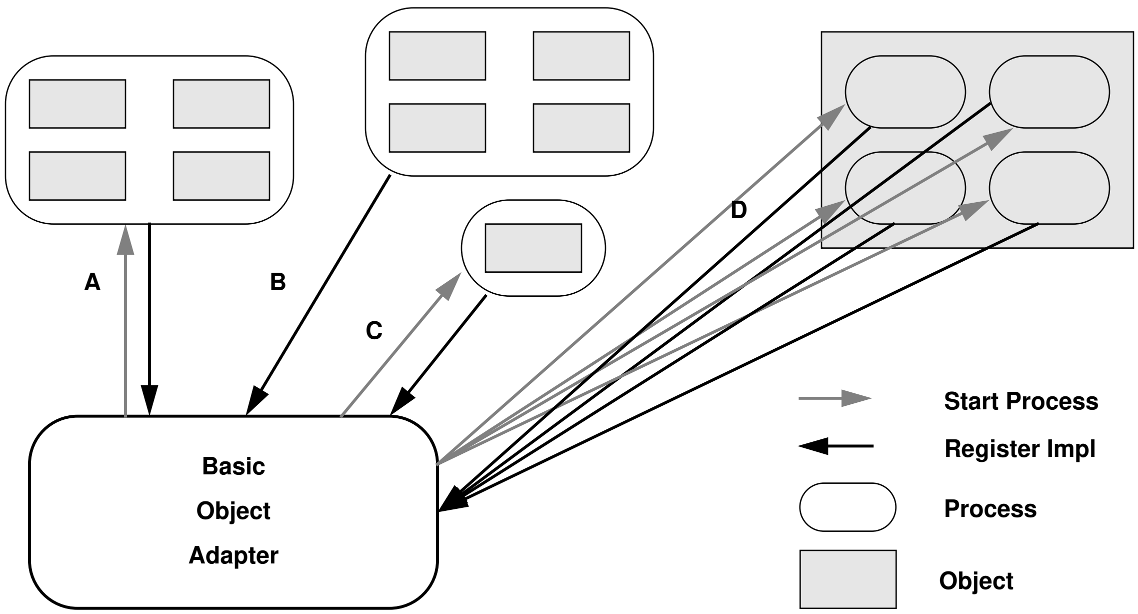
An *activation policy* describes the rules that a given implementation follows when there are multiple objects or implementations active. There are four policies that all BOA implementations support for implementation activation:

- A *shared server* policy, where multiple active objects of a given implementation share the same server.
- An *unshared server* policy, where only one object of a given implementation at a time can be active in one server.
- A *server-per-method* policy, where each invocation of a method is implemented by a separate server being started, with the server terminating when the method completes.
- *Persistent server* policy, where the server is activated by something outside the BOA. The server nonetheless must register with the BOA to receive invocations. A persistent server is assumed to be shared by multiple active objects.

These kinds of implementation activation are illustrated in FIG. 14 on page 153. Case A is a shared server, where the BOA starts a process which then registers itself with the BOA. Case B is the case of a persistent server, which is very similar but just registers itself with the BOA, without the BOA having had to start a process. An unshared server is illustrated in case C, where the process started by the BOA can only hold one object; the server-per-method policy in case D causes each method invocation to be done by starting a process.

2. The latter is for per-object servers.

FIG. 14 Implementation Activation Policies



9.2.2.1 Shared Server Activation Policy

In a shared server, multiple objects may be implemented by the same program. This is likely to be the most common kind of server. The server is activated the first time a request is performed on any object implemented by that server. When the server has initialized itself, it notifies the BOA that it is ready by calling **impl_is_ready**. Subsequently, the BOA will deliver requests or object activations for any objects implemented by that server. The server remains active and will receive requests until it calls **deactivate_impl**. The BOA will not activate another server for that implementation if one is active.

Before the first request is delivered for a particular object, the object activate routine of the server is called. An object remains active as long as its server is active, unless the server calls **deactivate_obj** for that object.

9.2.2.2 Unshared Server Activation Policy

In an unshared server, each object is implemented in a different server. This kind of server is convenient if a object is intended to encapsulate an application or if the server requires exclusive access to a resource such as a printer. A new server is activated the first time a request is performed on the object. When the server has initialized itself, it notifies the BOA that it is ready by calling **obj_is_ready**. Subsequently, the BOA will deliver requests for that object. The server remains active and will receive requests until it calls **deactivate_obj**.

A new server is started whenever a request is made for an object that is not yet active, even if a server for another object with the same implementation is active.

9.2.2.3 Server-per-Method Activation Policy

Under the server-per-method policy, a new server is always started each time a request is made. The server runs only for the duration of the particular method. Several servers for the same object or even the same method of the same object may be active simultaneously. Because a new server is started for each request, it is not necessary for the implementation to notify the BOA when an object is ready or deactivated.

The BOA activates an implementation for each request, whether or not another request for that operation, object, or implementation is active at the same time.

9.2.2.4 Persistent Server Activation Policy

Persistent servers are those servers which are activated by means outside the BOA. Such implementations notify the BOA that they are available using the **impl_is_ready** operation. Once the BOA knows about a persistent server, it treats the server as a shared server, sending it activations for individual objects and method calls. If no implementation is ready when a request arrives, an error is returned for that request.

9.2.3 Generation and Interpretation of Object References

Object references are generated by the BOA using the ORB Core when requested by an implementation. The BOA and the ORB Core work together to associate some information with a particular object reference. This information is later provided to the implementation upon the activation of an object. Note that this is the only information an implementation may use portably to distinguish different object references. The BOA operation used to create a new object reference is:

```
Object create (                                              // PIDL
    in ReferenceData          id,
    in InterfaceDef           intf,
    in ImplementationDef      impl
);
```

The **id** is immutable identification information, chosen by the implementation at object creation time, and never changed during the lifetime of the object. The **intf** is the Interface Repository object that specifies the complete set of interfaces implemented by the object. The **impl** is the Implementation Repository object that specifies the implementation to be used for the object.

A typical implementation will use the **id** value to distinguish different objects, but it is free to use it in any way it chooses or to assign the same value to different object references. Two object references created with the same parameters are *not* the same object reference as far as the ORB is concerned, although the implementation may or may not treat them as references to the same object. Note that the object reference itself is opaque and may be different for different ORBs, but the **id** value is available portably in all ORBs. Only the implementation can normally interpret the **id** value. The operation to get the **id** is a BOA operation:

```
ReferenceData get_id (in Object obj);                                    // PIDL
```

It is possible for the implementation associated with an object reference to be changed. This will cause subsequent requests to be handled according to the information in the new implementation. The operation to set the implementation is a BOA operation:

```
void change_implementation (                                             // PIDL
    in Object               obj,
    in ImplementationDef    impl
);
```

NOTE *Care must be taken in order to change the implementation after the object has been created. There are issues of synchronization with activation, security, and whether or not the new implementation is prepared to handle requests for that object. The* **change_implementation** *operation affects all copies of that particular object reference.*

If an object reference is copied, all copies have the same **id**, **intf**, and **impl.**

An implementation is allowed to dispose of an object it has created by asking the BOA to invalidate the object reference. The implementation is responsible for deallocating all other information about the object. After a **dispose** is done, the ORB Core and BOA act as if the object had never been created, and attempts to issue requests on any existing object references for that object will fail.

```
void dispose (in Object obj);                                            // PIDL
```

Note that all of the operations on object references in this section may be done whether or not the object is active.

9.2.4 Authentication and Access Control

The BOA does not enforce any specific style of security management. It guarantees that for every method invocation (or object activation) it will identify the principal on whose behalf the request is performed. The object implementation can obtain this principal by the operation:

```
Principal get_principal (                                   // PIDL
      in Object             obj,
      in Environment        ev
);
```

The **obj** parameter is the object reference passed to the method. If another object is used the result is undefined. The **ev** parameter is the language-mapping-specific request environment passed to the method.

The meaning of the principal depends on the security environment that the implementation is running in. The decision of whether or not to permit a particular operation is left up to the implementation. Typically, an implementation will associate access rights with particular objects and principals, and will examine those access rights to determine if the principal making the request has the privileges required by the particular method. An implementation could store a reference to the access control information for an object in the **id** for the object.

9.2.5 Persistent Storage

Objects (or, more precisely, object references) are made persistent by the BOA and the ORB Core, in that a client that has an object reference can use it at any time without warning, even if the implementation has been deactivated or the system has been restarted. Although the ORB Core and BOA maintain the persistence of object references, the implementation must participate in keeping any data outside the ORB Core and BOA persistent.

Toward this end, the BOA provides a small amount of storage for an object in the **id** value. In most cases, this storage is insufficient and inconvenient for the complete state of the object. Instead, the implementation provides and manages that storage, using the **id** value to locate the actual storage. For example, the **id** value might contain the name of a file, or a key for a database system that holds the persistent state.

9.3 C Language Mapping for Object Implementations

Different programming languages may provide access to the basic ORB functionality in different ways. Most of the issues of language mapping apply to all operations on all interfaces, and address such questions as the programming language view of the object reference, conventions for calling stubs and being called by skeletons, means of passing exception information, etc. There are a few details that apply specifically to the object adapter, such as how the implementation methods are connected to the skeleton.

9.3.1 Operation-specific details

Chapter 5 defines most of the details of naming of parameter types and parameter passing conventions. Generally, for those parameters that are operation-specific, the method implementing the operation appears to receive the same values that would be passed to the stubs.

9.3.2 Method signatures

With the BOA, implementation methods have signatures that are identical to the stubs.

If the following interface is defined in IDL:

```
interface example4 {
    long op5(in long arg6);
};
```

a method for the **op5** routine must have the following function signature:

```
long example4_op5(                                          /* C */
   example4       object,
   Environment    *ev,
   long           arg6
);
```

The **object** parameter is the object reference that was invoked. The method can identify which object was intended by using the **get_id** BOA operation. The **ev** parameter is used for authentication on the **get_principal** BOA operation, and is used for indicating exceptions.

The method terminates successfully by executing a **return** statement returning the declared operation value. Prior to returning the result of a successful invocation, the method code must assign legal values to all **out** and **inout** parameters.

The method terminates with an error by executing the **set_exception** BOA operation prior to executing a **return** statement. The **set_exception** operation has the following C language definition:

```
void BOA_set_exception (                                    /* C */
   Object            boa,
   Environment       *ev,
   exception_type    major,
   char              *exceptname,
   void              *param
);
```

The **ev** parameter is the environment parameter passed into the method. The caller must supply a value for the major parameter. The value of the major parameter constrains the other parameters in the call as follows:

- If the **major** parameter has the value NO_EXCEPTION, then it specifies that this is a normal outcome to the operation. In this case, both **exceptname** and **param** must be NULL. Note that it is *not* necessary to invoke **set_exception()** to indicate a normal outcome; it is the default behavior if the method simply returns.
- For any other value of **major** it specifies either a user-defined or standard exception. The **exceptname** parameter is a string representing the exception type identifier. If the exception is declared to take parameters, the **param** parameter must be the address of a struct containing the parameters according to the C language mapping, coerced to a **void ***; if the exception takes no parameters, **param** must be NULL.

When raising an exception, the method code is *not* required to assign legal values to any **out** or **inout** parameters. Due to restrictions in C, it must return a legal function value.

9.3.3 Binding Methods to Skeletons

It is not specified as part of the language mapping how the skeletons are connected to the methods. Different means will be used in different environments, for example, the skeletons may make references to the methods that are resolved by the linker or there may be a system-dependent call done at program startup to specify the location of the methods.

9.3.4 BOA and ORB routines

The operations on the BOA defined earlier in this chapter and the operations on the ORB defined in Chapter 8 are used as if they had the IDL definitions described in the document, and then mapped in the usual way with the C language mapping.

For example, the **string_to_object** ORB operation has the following signature:

```
Object ORB_string_to_object (                                   /* C */
   Object          orb,
   Environment     *ev,
   char            *objectstring
);
```

The **create** BOA operation has the following signature:

```
Object BOA_create (                                              /* C */
    Object                      boa,
    Environment                 *ev,
    ReferenceData               *id,
    InterfaceDef                intf,
    ImplementationDef           impl
);
```

Although in each example, we are using an “object” that is special (an ORB, an object adapter, or an object reference), the method name is generated as **interface_operation** in the same way as ordinary objects. Also, the signature contains an **Environment** parameter for error indications.

In the first two cases, the signature calls for an object reference to represent the particular ORB or object adapter being manipulated. Programs may obtain these objects in a variety of ways, for example, in a global variable before program startup if there is only one ORB or BOA that makes sense, or by obtaining them from a name service if more than one is available. In the third case, the object reference being operated on is specified as the first parameter.

Following the same procedure, the C language binding for the remainder of the ORB, BOA, and object reference operations may be determined.

10 Interoperability

It is an explicit goal of the Common ORB Architecture to allow interoperation between different object systems and ORBs. The large diversity of ORB implementation techniques means that a single strategy or technology for interoperation would be infeasible. However, there is substantial experience in the industry in connecting networks with different protocols, and we look to those working examples that are in everyday use for the model of how to connect ORBS.

In general, there is no single protocol that can meet everyone's needs, and there is no single means to interoperate between two different protocols. There are many environments in which multiple protocols coexist, and there are ways to bridge between environments that share no protocols. These same truths will hold for ORBs as well.

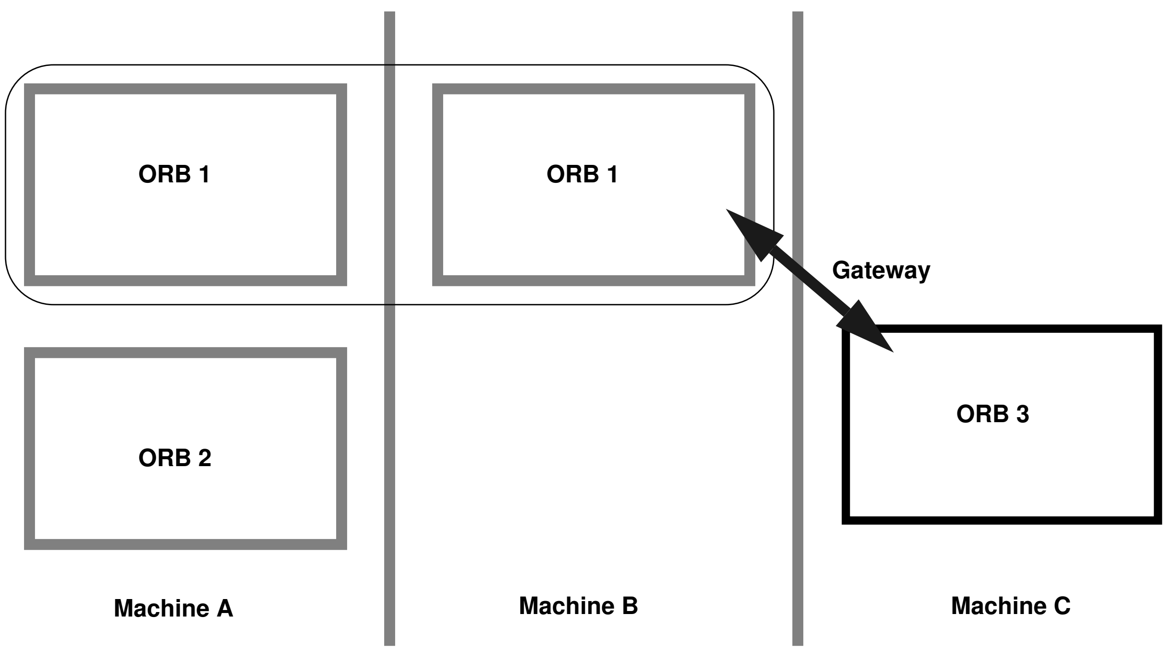
The primary requirement to allow convenient interoperation is to have a higher-level model that spans the differences. In the case of the ORB, there is an obvious higher-level model—IDL-defined object-oriented invocation. Because IDL is defined in an ORB-independent way, and because clients and object implementations can be built in an ORB-independent way, it is possible for a particular request to pass through multiple ORBs, preserving the invocation semantics transparent to clients and implementations.

10.1 The Organization of Multiple ORBs

FIG. 15 on page 162 shows three possible scenarios in which multiple ORBs coexist (although we will describe the first scenario as a single ORB).

1. ORB 1 is implemented on both Machine A and Machine B. Both implementations use the same object references and communication mechanism, and an object reference can be freely passed from Machine A to Machine B. We actually consider this case to be a single ORB implemented on two machines since no transformation is needed to move object references from one machine to the other.
2. On Machine A, the same client may have some objects implemented by ORB 1 and some by ORB 2. It is thus possible to an invoke an object reference in one ORB and pass as a parameter an object reference from another ORB. In any particular computing environment, an ORB must be able to distinguish its own object references from others', and must be able to pass other ORB's object references as parameters.
3. Between Machine B and Machine C, there are no common ORBs. In order to pass (and subsequently invoke) objects between Machine B and Machine C, it is necessary to construct a gateway to translate object references and requests in one ORB to object references and requests in the other.

FIG. 15 Multiple ORBs



There are many possible ways to connect two ORBS together, but they tend to fall into two categories: embedding of object references and protocol translation. Another technique is to allow object implementations to move objects between ORBs.

10.1.1 Reference Embedding

With reference embedding, an object in one ORB appears to be an object in a second ORB. An invocation on the object in the second ORB arrives at an implementation whose job it is to perform an invocation in the first ORB. On Machine A in FIG. 15 on page 162 an object implemented using ORB 1 might be made available in ORB 2 by creating an implementation in ORB 2 that, when invoked, simply invokes the corresponding object in ORB 1. A common use for reference embedding is when one ORB is a library ORB or other optimized implementation that cannot be accessed remotely. By embedding those objects that must be accessed remotely, most of the benefits of the optimized ORB can be had without sacrificing generality.

10.1.2 Protocol Translation

When two ORBs differ in their implementation details but have similar functionality, it will often be possible to translate requests in one ORB to be requests in the other ORB. For example, two RPC-based ORBs might differ in their object reference representation and packet formats, but otherwise present the same semantics. If it is possible to map object references in one ORB into object references in the other domain, and translate packets from one format to the other, a gateway could be constructed to pass requests back and forth.

10.1.3 Alternate ORBs

An object implementation implicitly chooses an ORB when it binds to a particular object adapter. If a machine supports multiple ORBs and the same object adapter interface is available on more than one ORB, an object implementation may choose to make the same object available through multiple ORBs. Because the object adapter interface is the same in each case, few changes would be necessary to the object implementation code to support multiple ORBs. However, the scope and performance of the different ORBs might be quite different.

In this case, an object implementation might generate object references in different ORBs. The interface to the object could define operations that would allow a client to obtain an equivalent object reference in a different ORB.

11 Glossary

activation — Preparing an object to execute an operation. For example, copying the persistent form of methods and stored data into an executable address space to allow execution of the methods on the stored data.

adapter — Same as object adapter.

attribute — An identifiable association between an object and a value. An attribute **A** is made visible to clients as a pair of operations: **get_A** and **set_A**. Readonly attributes only generate a **get** operation.

basic object adapter — The object adapter described in Chapter 9.

behavior — The observable effects of an object performing the requested operation including its results)binding. See language binding, dynamic invocation, static invocation, or method resolution for alternatives.

class — See interface and implementation for alternatives.

client The code or process that invokes an operation on an object.

context object A collection of name-value pairs that provides environmental or user-preference information. See Chapter 6.

CORBA Common Object Request Broker Architecture.

data type A categorization of values operation arguments), typically covering both behavior and representation. I.e., the traditional non-OO) programming language notion of type.

deactivation The opposite of activation.

deferred synchronous request A request where the client does not wait for completion of the request, but does intend to accept results later. Contrast with synchronous request and one-way request.

dynamic invocation Constructing and issuing a request whose signature is possibly not known until runtime.

externalized object reference An object reference expressed as an ORB-specific string. Suitable for storage in files or other external media.

implementation A definition that provides the information needed to create an object and allow the object to participate in providing an appropriate set of services. An implementation typically includes a description of the data structure used to represent the core state associated with an object, as well as definitions of the methods that access that data structure. It will also typically include information about the intended interface of the object.

implementation definition language A notation for describing implementations. The implementation definition language is currently beyond the scope of the ORB standard. It may contain vendor-specific and adapter-specific notations.

implementation inheritance The construction of an implementation by incremental modification of other implementations. The ORB does not provide implementation inheritance. Implementation inheritance may be provided by higher level tools.

implementation object An object that serves as an implementation definition. Implementation objects reside in an implementation repository.

implementation repository A storage place for object implementation information.

inheritance The construction of a definition by incremental modification of other definitions. See interface and implementation inheritance.

instance An object is an instance of an interface if it provides the operations signatures, semantics) specified by that interface. An object is an instance of an implementation if its behavior is provided by that implementation.

interface A listing of the operations and attributes that an object provides. This includes the signatures of the operations, and the types of the attributes. An interface definition ideally includes the semantics as well. An object *satisfies* an interface if it can be specified as the target object in each potential request described by the interface.

interface inheritance The construction of an interface by incremental modification of other interfaces. The IDL language provides interface inheritance.

interface object An object that serves to describe an interface. Interface objects reside in an interface repository.

interface repository A storage place for interface information.

interface type A type satisfied by any object that satisfies a particular interface.

interoperability The ability for two or more ORBs to cooperate to deliver requests to the proper object. Interoperating ORBs appear to a client to be a single ORB.

language binding or **mapping** The means and conventions by which a programmer writing in a specific programming language accesses ORB capabilities.

method An implementation of an operation. Code that may be executed to perform a requested service. Methods associated with an object may be structured into one or more programs.

method resolution The selection of the method to perform a requested operation.

multiple inheritance The construction of a definition by incremental modification of more than one other definition.

object — A combination of state and a set of methods that explicitly embodies an abstraction characterized by the behavior of relevant requests. An object is an instance of an implementation and an interface. An object models a real-world entity, and it is implemented as a computational entity that encapsulates state and operations (internally implemented as data and methods) and responds to requestor services.

object adapter — The ORB component which provides object reference, activation, and state related services to an object implementation. There may be different adapters provided for different kinds of implementations.

object creation — An event that causes the existence of an object that is distinct from any other object.

object destruction — An event that causes an object to cease to exist.

object implementation — Same as implementation.

object reference — A value that unambiguously identifies an object. Object references are never reused to identify another object.

objref — An abbreviation for object reference.

one-way request — A request where the client does not wait for completion of the request, nor does it intend to accept results. Contrast with deferred synchronous request and synchronous request.

operation — A service that can be requested. An operation has an associated signature, which may restrict which actual parameters are valid.

operation name — A name used in a request to identify an operation.

ORB — Object Request Broker. Provides the means by which clients make and receive requests and responses.

ORB core — The ORB component which moves a request from a client to the appropriate adapter for the target object.

parameter passing mode — Describes the direction of information flow for a operation parameter. The parameter passing modes are **IN**, **OUT**, and **INOUT**.

persistent object — An object that can survive the process or thread that created it. A persistent object exists until it is explicitly deleted.

referential integrity — The property ensuring that an object reference that exists in the state associated with an object reliably identifies a single object.

repository — See interface repository and implementation repository.

request — A client issues a request to cause a service to be performed. A request consists of an operation and zero or more actual parameters.

results — The information returned to the client, which may include values as well as status information indicating that exceptional conditions were raised in attempting to perform the requested service.

server — A process implementing one or more operations on one or more objects.

server object — An object providing response to a request for a service. A given object may be a client for some requests and a server for other requests.

signature — Defines the parameters of a given operation including their number order, data types, and passing mode; the results if any; and the possible outcomes (normal vs. exceptional) that might occur

single inheritance — The construction of a definition by incremental modification of one definition. Contrast with multiple inheritance.

skeleton — The object-interface-specific ORB component which assists an object adapter in passing requests to particular methods.

state — The time varying properties of an object that affect that object's behavior.

static invocation — Constructing a request at compile time. Calling an operation via a stub procedure.

stub — A local procedure corresponding to a single operation that invokes that operation when called.

synchronous request A request where the client pauses to wait for completion of the request. Contrast with deferred synchronous request and one-way request.

transient object An object whose existence is limited by the lifetime of the process or thread that created it.

type See data type and interface.

value Any entity that may be a possible actual parameter in a request. Values that serve to identify objects are called object references.

A Standard IDL Types

The IDL types listed in this appendix are available in all ORB implementations. IDL specifications that incorporate these types are therefore portable across ORB implementations. The types shown in TBL. 17 on page 171 are defined by the IDL language.

TBL. 17 Types Defined by IDL

Type	**Described In**
short	§4.7.1.1 on page 67
long	§4.7.1.1 on page 67
unsigned short	§4.7.1.1 on page 67
unsigned long	§4.7.1.1 on page 67
float	§4.7.1.2 on page 67
double	§4.7.1.2 on page 67
char	§4.7.1.3 on page 67
boolean	§4.7.1.4 on page 67
octet	§4.7.1.5 on page 67
struct	§4.7.2.1 on page 68
union	§4.7.2.2 on page 68
enum	§4.7.2.3 on page 70
sequence	§4.7.3.1 on page 70
string	§4.7.3.2 on page 71
array	§4.7.4.1 on page 71

TBL. 17 Types Defined by IDL (Continued)

Type	Described In
any	§4.7.1.6 on page 68
Object	§8.2 on page 145

TBL. 18 on page 172 lists the ORB pseudo-objects that should be available in any language mapping; in the C mapping, these definitions are contained in the file orb.h. Pseudo-objects cannot be invoked with the dynamic interface, and do not have object references. Those pseudo-objects that cannot be used as general arguments (passed as arguments in requests on real objects) are identified in the table. The definitions of pseudo-objects that can be used as general arguments are contained in the file orb.idl, and can be **#included** into IDL specifications.

TBL. 18 Pseudo-objects

Name	General Argument?	In orb.idl?	Described In
Environment	No	No	§5.19 on page 99
Request	No	No	§6.2 on page 108
Context	No	No	§6.5 on page 116
ORB	No	No	§8.1 on page 143
BOA	No	No	§9.2 on page 149
TypeCode	Yes	Yes	§7.6 on page 138
Principal	Yes	Yes	§9.2.4 on page 155
NamedValue	Yes	Yes	§6.1.1 on page 106
NVList	No	No	§6.1.1 on page 106

Types used with the Interface Repository are shown in TBL. 18 on page 172. They are contained in orb.idl.

TBL. 19 Interface Repository Types

Name	Type	Described In
Identifier	string	§7.5.1 on page 129
RepositoryId	string	§7.5.1 on page 129
OperationMode	enum	§7.5.6 on page 135
ParameterMode	enum	§7.5.7 on page 136
AttributeMode	enum	§7.5.5 on page 135
InterfaceDescription	struct	§7.5.4 on page 134

TBL. 19 Interface Repository Types (Continued)

Name	Type	Described In
OperationDescription	struct	§7.5.6 on page 135
AttributeDescription	struct	§7.5.5 on page 135
ParameterDescription	struct	§7.5.7 on page 136
RepositoryDescription	struct	§7.5.2 on page 132
ModuleDescription	struct	§7.5.3 on page 133
ConstDescription	struct	§7.5.9 on page 137
ExceptionDescription	struct	§7.5.10 on page 138
TypeDescription	struct	§7.5.8 on page 137
FullInterfaceDescription	struct	§7.5.4 on page 134
InterfaceDef	interface	§7.5.4 on page 134
OperationDef	interface	§7.5.6 on page 135
AttributeDef	interface	§7.5.5 on page 135
ParameterDef	interface	§7.5.7 on page 136
RepositoryDef	interface	§7.5.2 on page 132
ModuleDef	interface	§7.5.3 on page 133
TypeDef	interface	§7.5.8 on page 137
ConstDef	interface	§7.5.9 on page 137
ExceptionDef	interface	§7.5.10 on page 138
ImplementationDef	interface	§9.2.1 on page 151

The **any** type can be used to represent a variety of types of values. All ORB implementations must support at least the types listed in TBL. 20 on page 173 as **any**. ORB implementations may define more values for **any**.

TBL. 20 Minimal Types for Any

IDL Basic Types
Object
Any
TypeCode
NamedValue
user-defined interfaces
InterfaceDescription
OperationDescription
AttributeDescription

TBL. 20 Minimal Types for Any (Continued)

ParameterDescription
RepositoryDescription
ModuleDescription
ConstDescription
ExceptionDescription
TypeDescription
FullInterfaceDescription
sequences of the above