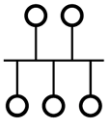


Orbix®

CORBA Programmer's Guide, Java

Version 6.1, December 2003

Making Software Work Together™

Updated: 28-May-2004

M 3 1 4 4

Contents

List of Figures

List of Tables

Preface

Orbix is a full implementation from IONA Technologies of the Common Object Request Broker Architecture (CORBA), as specified by the Object Management Group. Orbix complies with the following specifications:

- CORBA 2.3
- GIOP 1.2 (default), 1.1, and 1.0

Audience

The *CORBA Programmer's Guide* is intended to help you become familiar with Orbix, and show how to develop distributed applications using Orbix components. This guide assumes that you are familiar with programming in Java.

This guide does not discuss every API in great detail, but gives a general overview of the capabilities of the Orbix development kit and how various components fit together.

Organization of this guide

Read Chapter 1 for an overview of Orbix. Chapter 2 shows how you can use code-generation genies to build a distributed application quickly and easily. Chapter 3 describes in detail the basic steps in building client and server programs. Subsequent chapters expand on those steps by focusing on topics that are related to application development.

Additional resources

The IONA knowledge base (http://www.iona.com/support/knowledge_base/index.xml) contains helpful articles, written by IONA experts, about the Orbix and other products. You can access the knowledge base at the following location:

The IONA update center (http://www.iona.com/support/updates/index.xml) contains the latest releases and patches for IONA products:

If you need help with this or any other IONA products, contact IONA at support@iona.com. Comments on IONA documentation can be sent to docs-support@iona.com .

Typographical conventions

This guide uses the following typographical conventions:

`Constant width`	Constant width (courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `CORBA::Object` class. Constant width paragraphs represent code examples or information a system displays on the screen. For example: `#include <stdio.h>`
Italic	Italic words in normal text represent *emphasis* and *new terms*. Italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example: `% cd /users/`*`your_name`* **Note:** Some command examples may use angle brackets to represent variable values you must supply. This is an older convention that is replaced with *italic* words or characters.

Keying conventions

This guide may use the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, a prompt is not used.
%	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
#	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
>	The notation > represents the DOS or Windows command prompt.
. . . . . .	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.
[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices enclosed in { } (braces) in format and syntax descriptions.

CHAPTER 1

Introduction to Orbix

With Orbix, you can develop and deploy large-scale enterprise-wide CORBA systems in C++ and Java. Orbix has an advanced modular architecture that lets you configure and change functionality without modifying your application code, and a rich deployment architecture that lets you configure and manage a complex distributed system.

In this chapter

This chapter contains the following sections:

Why CORBA?

Overview

Today's enterprises need flexible, open information systems. Most enterprises must cope with a wide range of technologies, operating systems, hardware platforms, and programming languages. Each of these is good at some important business task; all of them must work together for the business to function.

The common object request broker architecture—CORBA—provides the foundation for flexible and open systems. It underlies some of the Internet's most successful e-business sites, and some of the world's most complex and demanding enterprise information systems.

What is CORBA?

CORBA is an open, standard solution for distributed object systems. You can use CORBA to describe your enterprise system in object-oriented terms, regardless of the platforms and technologies used to implement its different parts. CORBA objects communicate directly across a network using standard protocols, regardless of the programming languages used to create objects or the operating systems and platforms on which the objects run.

CORBA solutions are available for every common environment and are used to integrate applications written in C, C++, Java, Ada, Smalltalk, and COBOL, running on embedded systems, PCs, UNIX hosts, and mainframes. CORBA objects running in these environments can cooperate seamlessly. Through COMet, IONA's dynamic bridge between CORBA and COM, they can also interoperate with COM objects.

CORBA is widely available and offers an extensive infrastructure that supports all the features required by distributed business objects. This infrastructure includes important distributed services, such as transactions, security, and messaging.

Orbix

Orbix provides a CORBA development platform for building high-performance systems. Orbix's modular architecture supports the most demanding requirements for scalability, performance, and deployment flexibility. The Orbix architecture is also language-independent and can be

implemented in Java and C++. Orbix applications can interoperate via the standard IIOP protocol with applications built on any CORBA-compliant technology.

CORBA Objects

CORBA objects are abstract objects in a CORBA system that provide distributed object capability between applications in a network. Figure 1 shows that any part of a CORBA system can refer to the abstract CORBA object, but the object is only implemented in one place and time on some server of the system.

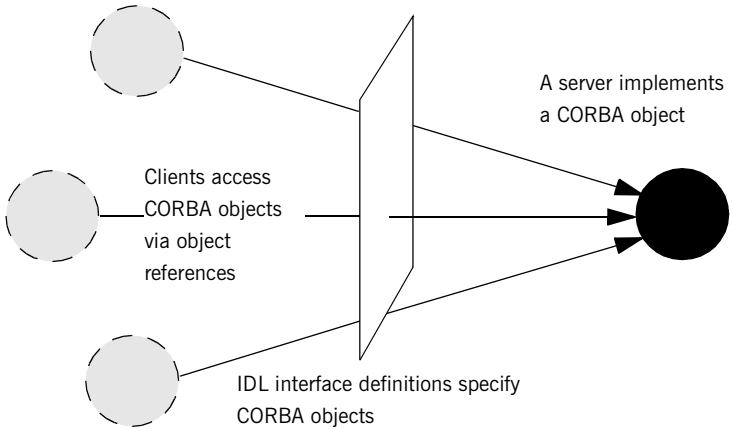


Figure 1: *The nature of abstract CORBA objects*

An *object reference* is used to identify, locate, and address a CORBA object. Clients use an object reference to invoke requests on a CORBA object. CORBA objects can be implemented by servers in any supported programming language, such as C++ or Java.

Although CORBA objects are implemented using standard programming languages, each CORBA object has a clearly-defined interface, specified in the *CORBA Interface Definition Language* (*IDL*). The *interface definition* specifies which member functions, data types, attributes, and exceptions are available to a client, without making any assumptions about an object's implementation.

With a few calls to an ORB's application programming interface (API), servers can make CORBA objects available to client programs in your network.

To call member functions on a CORBA object, a client programmer needs only to refer to the object's interface definition. Clients can call the member functions of a CORBA object using the normal syntax of the chosen programming language. The client does not need to know which programming language implements the object, the object's location on the network, or the operating system in which the object exists.

Using an IDL interface to separate an object's use from its implementation has several advantages. For example, you can change the programming language in which an object is implemented without affecting the clients that access the object. You can also make existing objects available across a network.

Object Request Broker

CORBA defines a standard architecture for object request brokers (ORB). An ORB is a software component that mediates the transfer of messages from a program to an object located on a remote network host. The ORB hides the underlying complexity of network communications from the programmer.

An ORB lets you create standard software objects whose member functions can be invoked by *client* programs located anywhere in your network. A program that contains instances of CORBA objects is often known as a *server*. However, the same program can serve at different times as a client and a server. For example, a server program might itself invoke calls on other server programs, and so relate to them as a client.

When a client invokes a member function on a CORBA object, the ORB intercepts the function call. As shown in Figure 2, the ORB redirects the function call across the network to the target object. The ORB then collects results from the function call and returns these to the client.

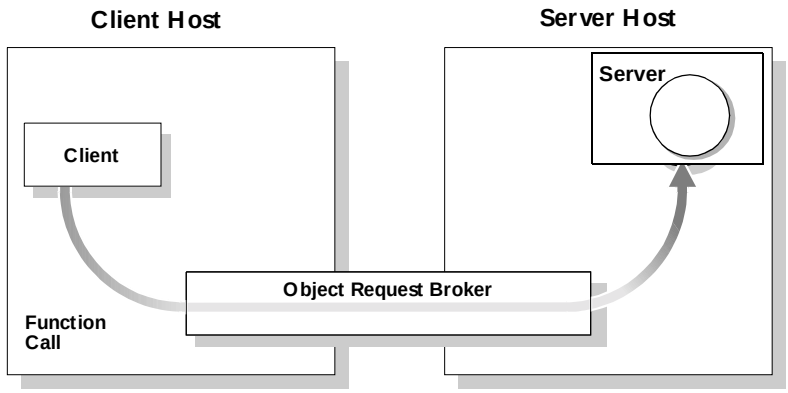


Figure 2: *The object request broker*

CORBA Application Basics

You start developing a CORBA application by defining interfaces to objects in your system in CORBA IDL. You compile these interfaces with an IDL compiler. An IDL compiler generates C++ or Java code from IDL definitions. This code includes *client stub code* with which you develop client programs, and *object skeleton code*, which you use to implement CORBA objects.

When a client calls a member function on a CORBA object, the call is transferred through the client stub code to the ORB. Because the implemented object is not located in the client's address space, CORBA objects are represented in client code by *proxy objects*.

A client invokes on object references that it obtains from the server process. The ORB then passes the function call through the object skeleton code to the target object.

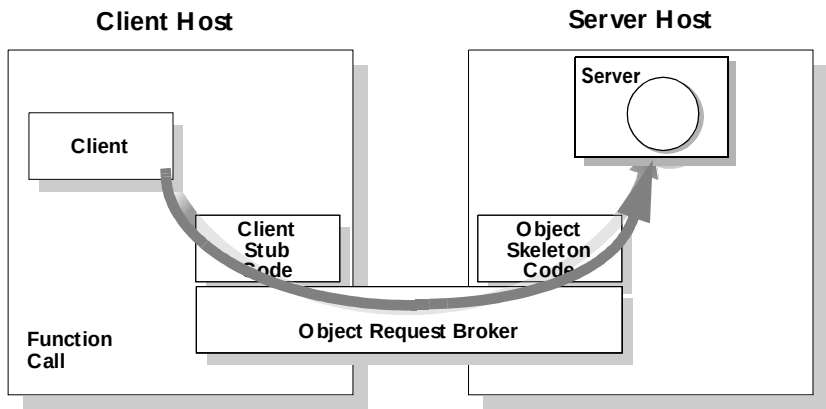


Figure 3: *Invoking on a CORBA object*

Servers and the Portable Object Adapter

Server processes act as containers for one or more *portable object adapters.* A portable object adapter, or POA, maps abstract CORBA objects to their actual implementations, or *servants*, as shown in Figure 4. Because the

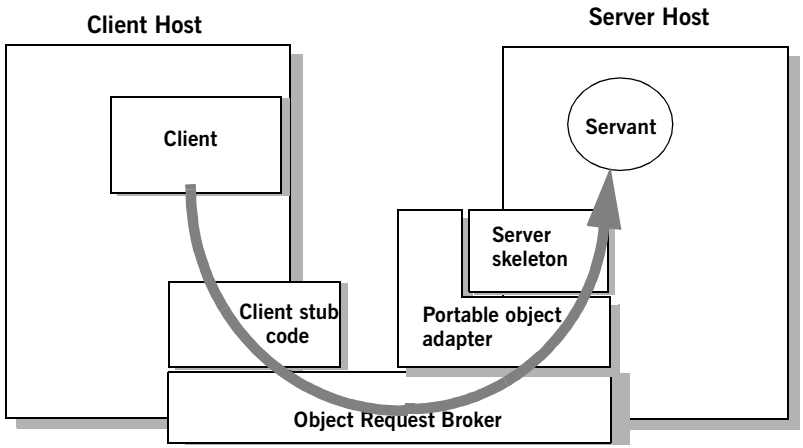


Figure 4: *The portable object adapter*

POA assumes responsibility for mapping servants to abstract CORBA objects, the way that you define or change an object's implementation is transparent to the rest of the application. By abstracting an object's identity from its implementation, a POA enables a server to be portable among different implementations.

Depending on the policies that you set on a POA, object-servant mappings can be static or dynamic. POA policies also determine whether object references are persistent or transient, and the threading model that it uses. In all cases, the policies that a POA uses to manage its objects are invisible to clients.

A server can have one or more nested POAs. Because each POA has its own set of policies, you can group objects logically or functionally among multiple POAs, where each POA is defined in a way that best accommodates the needs of the objects that it processes.

Orbix Plug-In Design

Orbix has a modular *plug-in* architecture. The ORB core supports abstract CORBA types and provides a plug-in framework. Support for concrete features like specific network protocols, encryption mechanisms, and database storage is packaged into plug-ins that can be loaded into the ORB based on runtime configuration settings.

Plug-ins

A plug-in is a code library that can be loaded into an Orbix application at runtime. A plug-in can contain any type of code; typically, it contains objects that register themselves with the ORB runtimes to add functionality.

Plug-ins can be linked directly with an application, loaded when an application starts up, or loaded on-demand while the application is running. This gives you the flexibility to choose precisely those ORB features that you actually need. Moreover, you can develop new features such as protocol support for direct ATM or HTTPNG. Because ORB features are *configured* into the application rather than *compiled* in, you can change your choices as your needs change without rewriting or recompiling applications.

For example, an application that uses the standard IIOP protocol can be reconfigured to use the secure SSL protocol simply by configuring a different transport plug-in. No one transport is inherent to the ORB core; you simply load the transport set that suits your application best. This architecture makes it easy for IONA to support additional transports in the future such as multicast or special purpose network protocols.

ORB core

The ORB core presents a uniform programming interface to the developer: *everything is a CORBA object*. This means that everything appears to be a local C++ or Java object within the process. In fact it might be a local object, or a remote object reached by some network protocol. It is the ORB's job to get application requests to the right objects no matter where they live.

To do its job, the ORB loads a collection of plug-ins as specified by ORB configuration settings—either on startup or on demand—as they are needed by the application. For remote objects, the ORB intercepts local function calls and turns them into CORBA *requests* that can be dispatched to a remote object.

In order to send a request on its way, the ORB core sets up a chain of *interceptors* to handle requests for each object. The ORB core neither knows nor cares what these interceptors do, it simply passes the request along the interceptor chain. The chain might be a single interceptor which sends the request with the standard IIOP protocol, or a collection of interceptors that add transaction information, encrypt the message and send it on a secure protocol such as SSL. All of this is transparent to the application, so you can change the protocol or services used by your application simply by configuring a different set of interceptors.

Development Tools

The CORBA developer's environment contains a number of facilities and features that help you and your development team be more productive.

Code generation toolkit

IONA provides a code generation toolkit that simplifies and streamlines the development effort. You only need to define your IDL interfaces; out-of-the box scripts generate a complete client/server application automatically from an IDL file.

The toolkit also can be useful for debugging: you can use an auto-generated server to debug your client, and vice versa. Advanced users can write code generation scripts to automate repetitive coding in a large application.

For more information about the code generation toolkit, refer to the *CORBA Code Generation Toolkit Guide*.

Multi-threading support

Orbix provides excellent support for multi-threaded applications. Orbix libraries are multi-threaded and thread-safe. Orbix servers use standard POA policies to enable multi-threading. The ORB creates a thread pool that automatically grows or shrinks depending on demand load. Thread pool size, growth and request queuing can be controlled by configuration settings without any coding.

Configuration and logging interfaces

Applications can store their own configuration information in Orbix configuration domains, taking advantage of the infrastructure for ORB configuration. CORBA interfaces provide access to configuration information in application code.

Applications can also take advantage of the Orbix logging subsystem, again using CORBA interfaces to log diagnostic messages. These messages are logged to log-stream objects that are registered with the ORB. Log streams for local output, file logging and system logging (Unix syslogd or Windows Event Service) are provided with Orbix. You can also implement your own log streams, which capture ORB and application diagnostics and send them to any destination you desire.

Portable interceptors

Portable interceptors allow an application to intervene in request handling. They can be used to log per-request information, or to add extra “hidden” data to requests in the form of GIOP service contexts—for example, transaction information or security credentials.

Orbix Application Deployment

Orbix provides a rich deployment environment designed for high scalability. You can create a *location domain* that spans any number of hosts across a network, and can be dynamically extended with new hosts. Centralized domain management allows servers and their objects to move among hosts within the domain without disturbing clients that use those objects. Orbix supports load balancing across object groups. A *configuration domain* provides the central control of configuration for an entire distributed application.

Orbix offers a rich deployment environment that lets you structure and control enterprise-wide distributed applications. Orbix provides central control of all applications within a common domain.

Location domains

A location domain is a collection of servers under the control of a single locator daemon. The locator daemon can manage servers on any number of hosts across a network. The locator daemon automatically activates remote servers through a stateless activator daemon that runs on the remote host.

The locator daemon also maintains the implementation repository, which is a database of available servers. The implementation repository keeps track of the servers available in a system and the hosts they run on. It also provides a central forwarding point for client requests. By combining these two functions, the locator lets you relocate servers from one host to another without disrupting client request processing. The locator redirects requests to the new location and transparently reconnects clients to the new server instance. Moving a server does not require updates to the naming service, trading service, or any other repository of object references.

The locator can monitor the state of health of servers and redirect clients in the event of a failure, or spread client load by redirecting clients to one of a group of servers.

Configuration domains

A configuration domain is a collection of applications under common administrative control. A configuration domain can contain multiple location domains.

Orbix supports two mechanisms to administer a configuration domain:

- During development, or for small-scale deployment, configuration can be stored in an ASCII text file, which is edited directly.
- For larger deployments, Orbix provides a distributed configuration server that enables centralized configuration for all applications spread across a network.

The configuration mechanism is loaded as a plug-in, so future configuration systems can be extended to load configuration from any source such as example HTTP or third-party configuration systems.

CORBA Features and Services

Orbix fully supports the latest CORBA specification, and in some cases anticipates features to be included in upcoming specifications.

Full CORBA 2.3 support and interoperability

All CORBA 2.3 IDL data types are fully supported, including:

- Extended precision numeric types for 64 bit integer and extended floating point calculations.
- Fixed point decimals for financial calculations.
- International character sets, including support for code-set negotiation where multiple character sets are available.
- Objects by value: you can define objects that are passed by value as well as the traditional pass-by-reference semantics of normal CORBA objects. This is particularly relevant in Java based systems, but also supported for C++ using object factories.

Orbix supports the most recent 1.2 revision of the CORBA standard General Inter-ORB Protocol (GIOP) and Internet Inter-ORB Protocol (IIOP), and also supports previous 1.1 and 1.0 revisions for backwards compatibility with applications based on other ORBs. Orbix is interoperable with any CORBA-compliant application that uses the standard IIOP protocol.

Orbix implements quality-of-service policies as specified in CORBA 3.0. Quality-of-service policies let you control how the ORB processes requests. For example, you can specify how quickly a client resumes processing after sending one-way requests.

Interoperable naming service and load balancing extensions

Orbix supports the interoperable naming service specification. This is a superset of the original CORBA naming service which adds some ease-of-use features and provides a standard URL format for CORBA object references to simplify configuration and administration of CORBA services.

The Orbix naming service also supports IONA-specific load-balancing extensions of OrbixNames 3. A group of objects can be registered against a single name; the naming service hands out references to clients so that the client load is spread across the group.

Object transaction service

Orbix includes the object transaction service (OTS) which is optimized for the common case where only a single resource (database) is involved in a transaction. Applications built against the single resource OTS can easily be reconfigured to use a full-blown OTS when it is available, since the interfaces are identical. With Orbix plug-in architecture, applications will not even need to be recompiled. For the many applications where transactions do not span multiple databases, the single-resource OTS will continue to be a highly efficient solution, compared to a full OTS that performs extensive logging to guarantee transaction integrity.

Event service

Orbix supports the CORBA event service specification, which defines a model for indirect communications between ORB applications. A client does not directly invoke an operation on an object in a server. Instead, the client sends an event that can be received by any number of objects. The sender of an event is called a *supplier*; the receivers are called *consumers*. An intermediary *event channel* takes care of forwarding events from suppliers to consumers.

Orbix supports both the *push* and *pull* model of event transfer, as defined in the CORBA event specification. Orbix performs event transfer using the *untyped* format, whereby events are based on a standard operation call that takes a generic parameter of type any.

SSL/TLS

Orbix SSL/TLS provides data security for applications that communicate across networks by ensuring authentication, privacy, and integrity features for communications across TCP/IP connections.

TLS is a transport layer security protocol layered between application protocols and TCP/IP, and can be used for communication by all Orbix SSL/TLS components and applications.

COMet

OrbixCOMet provides a high performance dynamic bridge that enables transparent communication between COM/Automation clients and CORBA servers.

OrbixCOMet is designed to give COM programmers—who use tools such as Visual C++, Visual Basic, PowerBuilder, Delphi, or Active Server Pages on the Windows desktop—easy access to CORBA applications running on

Windows, UNIX, or OS/390 environments. COM programmers can use the tools familiar to them to build heterogeneous systems that use both COM and CORBA components within a COM environment.

Dynamic type support: interface repository and dynany

Orbix has full support for handling data values that are not known at compile time. The interface repository stores information about all CORBA types known to the system and can be queried at runtime. Clients can construct requests based on runtime type information using the dynamic invocation interface (DII), and servers can implement "universal" objects that can implement any interface at run time with the dynamic skeleton interface (DSI).

Although all of these features have been available since early releases of the CORBA specification, they are incomplete without the addition of the DynAny interface. This interface allows clients and servers to interpret or construct values based purely on runtime information, without any compiled-in data types.

These features are ideal for building generic object browsers, type repositories, or protocol gateways that map CORBA requests into another object protocol.

CHAPTER 2

Getting Started with Orbix

You can use the CORBA Code Generation Toolkit to develop an Orbix application quickly.

Given a user-defined IDL interface, the toolkit generates the bulk of the client and server application code, including build files. You then complete the distributed application by filling in the missing business logic.

In this chapter

This chapter contains the following sections:

Creating a Configuration Domain

Overview

This section describes how to create a simple configuration domain, `simple`, which is required for running basic demonstrations. This domain deploys a minimal set of Orbix services.

Prerequisites

Before creating a configuration domain, the following prerequisites must be satisfied:

- Orbix is installed.
- Some basic system variables are set up (in particular, the `IT_PRODUCT_DIR`, `IT_LICENSE_FILE`, and `PATH` variables).

Fore more details, please consult the *Installation Guide*.

Licensing

The location of the license file, `licenses.txt`, is specified by the `IT_LICENSE_FILE` system variable. If this system variable is not already set in your environment, you can set it now.

Steps

To create a configuration domain, `simple`, perform the following steps:

1. Run itconfigure.
2. Specify the license location.
3. Choose expert mode and specify domain settings.
4. Specify services settings.
5. Review the summary window.
6. Finish configuration.

Run itconfigure

To begin creating a new configuration domain, enter `itconfigure` at a command prompt. An **Introduction** window appears, as shown in Figure 5.

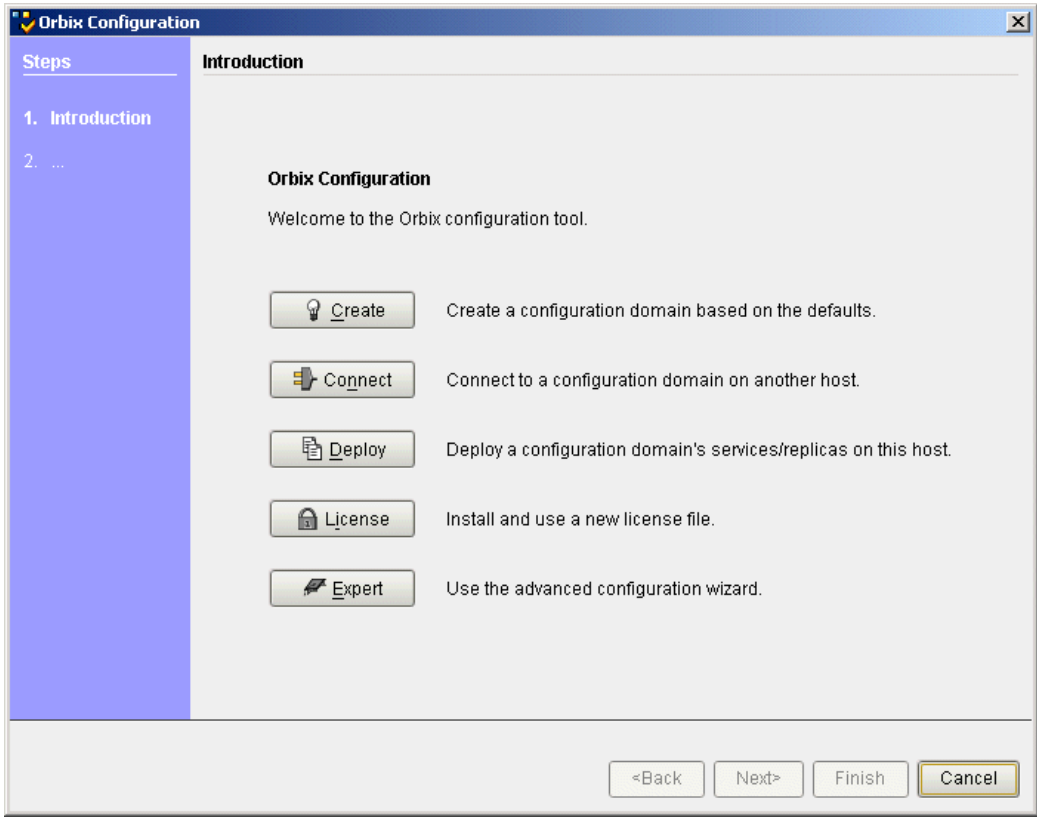


Figure 5: *The itconfigure Introduction Window*

Specify the license location

If you have not already specified the license location by setting the `IT_LICENSE_FILE` environment variable (see "Licensing" on page 20), specify the location now by clicking the **License** button on the **Introduction** window (Figure 5 on page 21).

A License dialog box appears, as shown in Figure 6. Enter the license file location in the **License File** text field or use the **Browse** button to select the license file, then click **OK**.

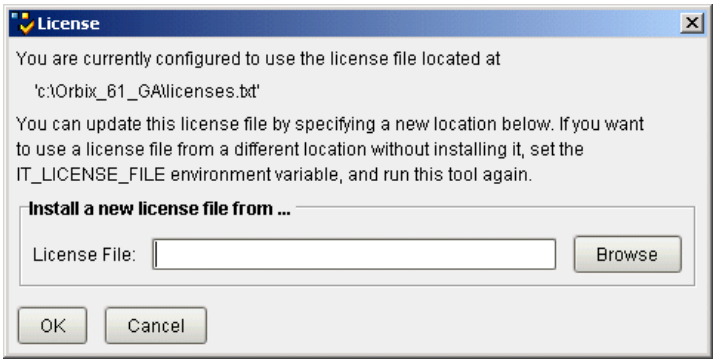


Figure 6: *The License Dialog Box*

Choose expert mode and specify domain settings

From the **Introduction** window (Figure 5 on page 21), click **Expert** to begin creating a configuration domain in expert mode. A **Domain Settings** window appears, as shown in Figure 7.

In the **Domain Name** text field, type `simple`. Select the **File Based Domain** option.

Make sure that the **Allow Insecure Communication** option is selected and the **Allow Secure Communication** option is unselected.

Click **Next>** to continue.

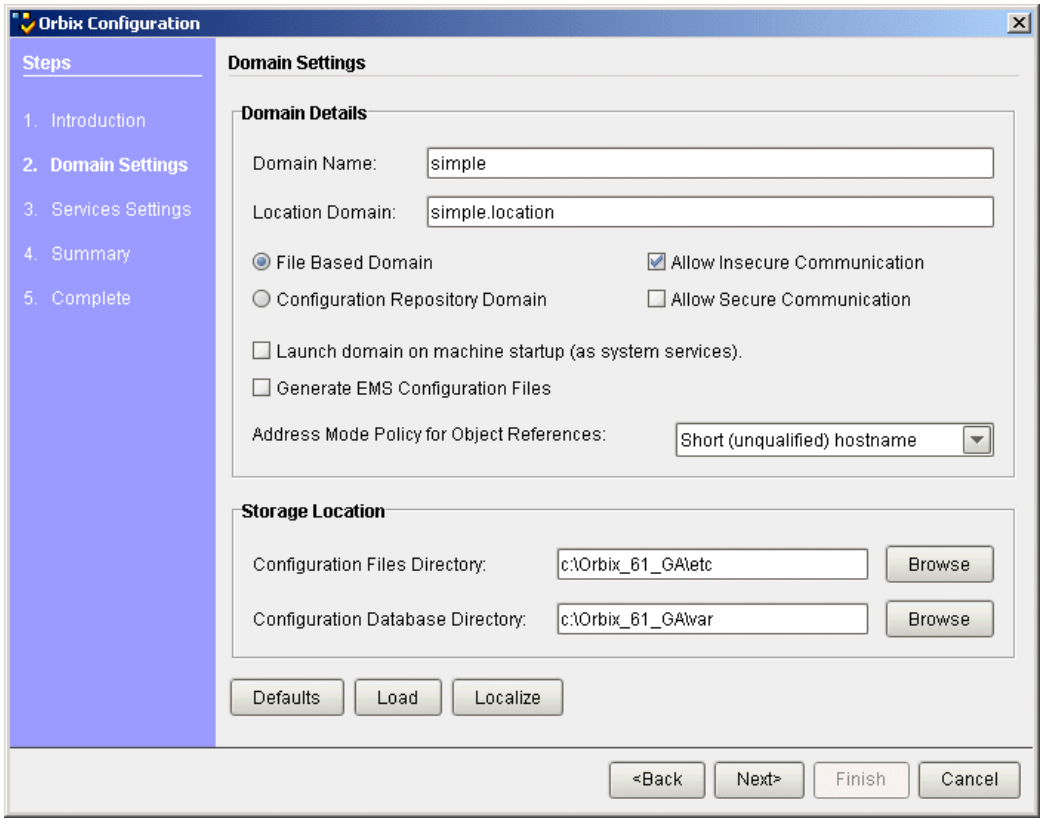


Figure 7: *The itconfigure Domain Settings Window*

Specify services settings

A **Services Settings** window appears, as shown in Figure 8.

In the **Services Settings** window, select the following services and components for inclusion in the configuration domain: **Location**, **Node daemon**, **Management**, **Distributed Transaction**, **CORBA Interface Repository**, **CORBA Naming**, and **Demos**.

Click **Next>** to continue.

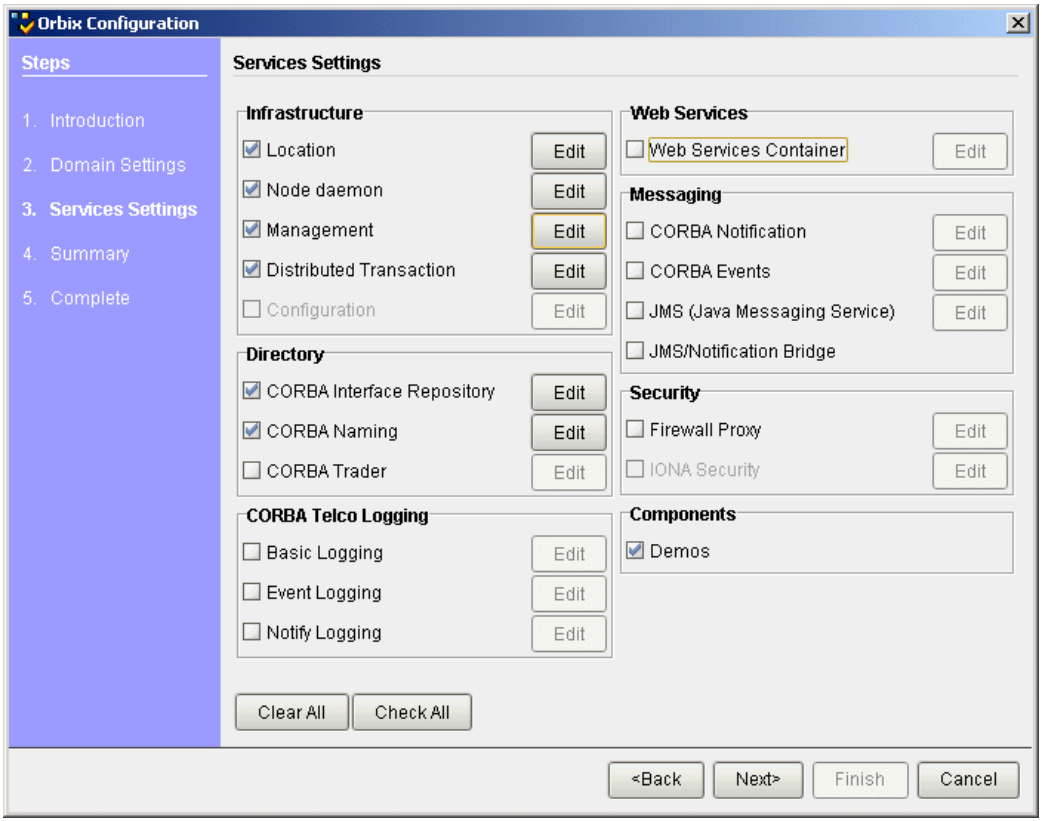


Figure 8: *The itconfigure Services Settings Window*

Review the summary window

You now have the opportunity to review the configuration settings in the **Summary** window, Figure 9. If necessary, you can use the **<Back** button to make corrections.

Click **Next>** to create the configuration domain and progress to the next window.

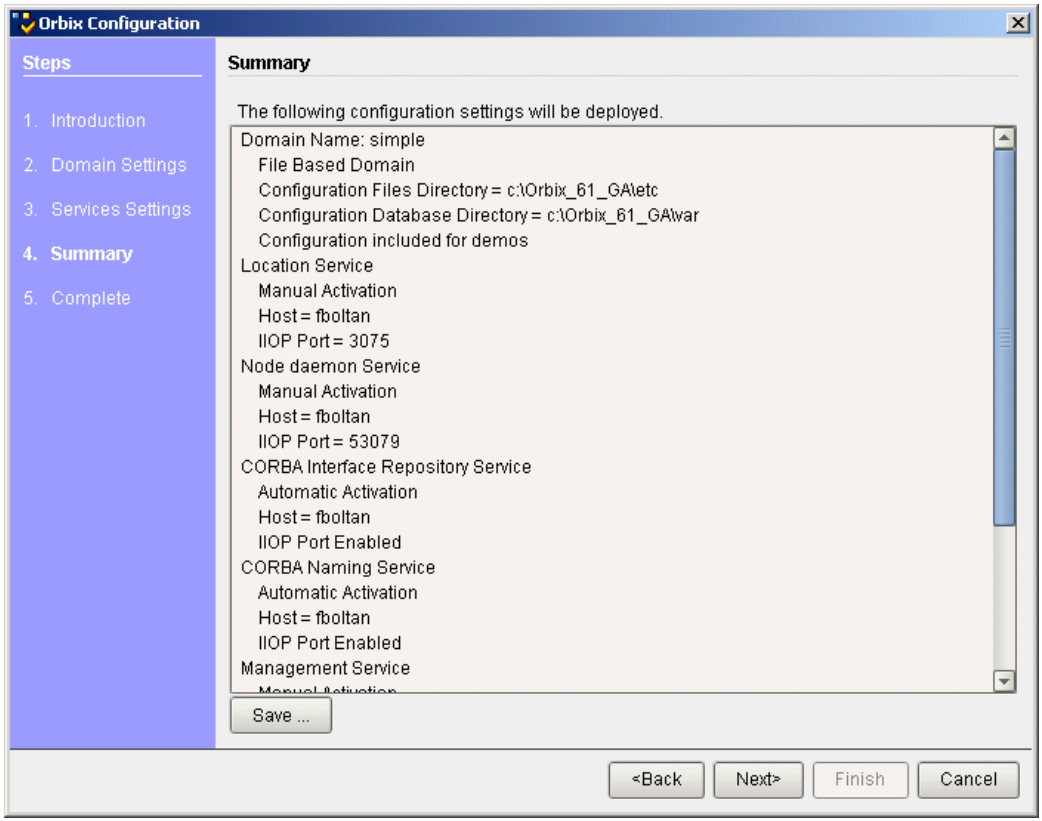


Figure 9: *The itconfigure Summary Window*

Finish configuration

The `itconfigure` utility now creates and deploys the `simple` configuration domain, writing files into the *OrbixInstallDir*`/etc/bin`, *OrbixInstallDir*`/etc/domain`, *OrbixInstallDir*`/etc/log`, and *OrbixInstallDir*`/var` directories.

If the configuration domain is created successfully, you should see a **Complete** window with a message similar to that shown in Figure 10.

Click **Finish** to quit the `itconfigure` utility.

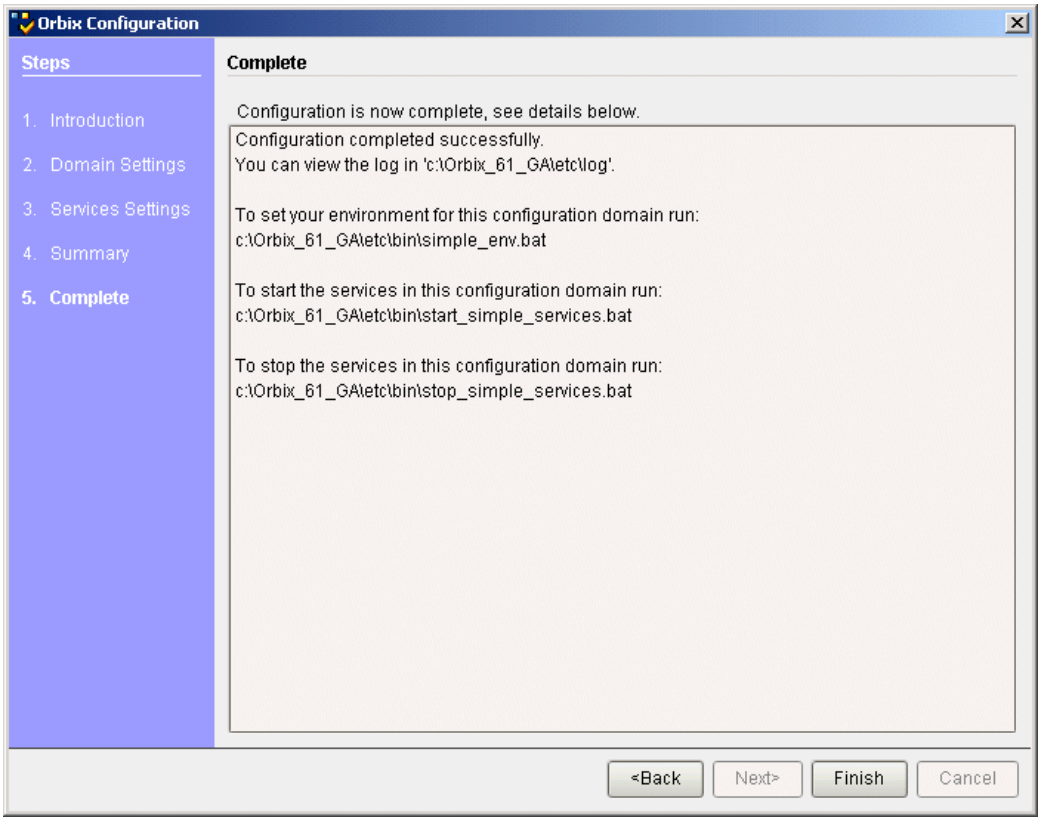


Figure 10: *Finishing Configuration*

Setting the Orbix Environment

Prerequisites

Before proceeding with the demonstration in this chapter you need to ensure:

- The CORBA developer's kit is installed on your host.
- Orbix is configured to run on your host platform.
- Your Java development kit (JDK) is configured to use the Orbix ORB runtime (see "Setting ORB Properties for the Orbix ORB" on page 28).

The *Administrator's Guide* contains more information on Orbix configuration, and details of Orbix command line utilities.

Setting the Domain

The scripts that set the Orbix environment are associated with a particular *domain*, which is the basic unit of Orbix configuration. Consult the *Installation Guide*, and the *Administrator's Guide* for further details on configuring your environment.

To set the Orbix environment associated with the *domain-name* domain, enter:

Windows

```
> set JAVA_HOME=YourJdkDir
> config-dir\etc\bin\domain-name_env.bat
```

UNIX

```
% JAVA_HOME=YourJdkDir ; export JAVA_HOME
% . config-dir/etc/bin/domain-name_env
```

YourJdkDir is the root directory of the Java development kit that you want to use with Orbix. See the *Installation Guide* for details of supported Java platforms.

config-dir is the root directory where the Appliation Server Platform stores its configuration information. You specify this directory while configuring your domain. *domain-name* is the name of a configuration domain.

Setting ORB Properties for the Orbix ORB

SUN's Java development kit (JDK) comes with a built-in ORB runtime that is used by default. However, you cannot use SUN's ORB runtime with Orbix applications. You must configure the JDK to use the Orbix ORB runtime instead by setting system properties `org.omg.CORBA.ORBClass` and `org.omg.CORBA.ORBSingletonClass` to the appropriate values. You can set the ORB properties in one of the following ways:

- Using the iona.properties file
- Using Java interpreter arguments

Using the iona.properties file

Setting system properties `org.omg.CORBA.ORBClass` and `org.omg.CORBA.ORBSingletonClass` in the `iona.properties` file is the preferred way to configure your JDK to use the Orbix ORB runtime.

Location of the iona.properties file

The `iona.properties` file is located in the *`JDKHome`*`/jre/lib` directory, where *`JDKHome`* is the JDK root directory.

Contents of the iona.properties file

The `iona.properties` file should contain the following two lines of text:

```
org.omg.CORBA.ORBClass=com.iona.corba.art.artimpl.ORBImpl
org.omg.CORBA.ORBSingletonClass=
    com.iona.corba.art.artimpl.ORBSingleton
```

The first line sets `org.omg.CORBA.ORBClass` to the name of a class that implements `org.omg.CORBA.ORB`.

The second line sets `org.omg.CORBA.ORBSingletonClass` to the name of a class that implements the static ORB instance returned from `org.omg.CORBA.ORB.init()` (taking no arguments).

WARNING: By setting system properties `org.omg.CORBA.ORBClass` and `org.omg.CORBA.ORBSingletonClass` in the `iona.properties` file, as detailed above, you effectively specify the Orbix ORB classes as the ORB runtime for the JDK. This might affect other applications that use the same JDK but want to use different ORB classes—if this is the case, you should consider using one of the alternative mechanisms for setting ORB properties, given in the following sub-sections.

Using Java interpreter arguments

You can use the `-D`*`property_name`*`=property_value` option on the Java Interpreter to specify the `org.omg.CORBA.ORBClass` and `org.omg.CORBA.ORBSingletonClass` properties. For example, to set the ORB properties for an `orbix_app` Orbix application:

```
java -Dorg.omg.CORBA.ORB=com.iona.corba.art.artimpl.ORBImpl\
     -Dorg.omg.CORBA.ORBSingletonClass=\
     com.iona.corba.art.artimpl.ORBSingleton orbix_app
```

Hello World Example

This chapter shows how to create, build, and run a complete client/server demonstration with the help of the CORBA code generation toolkit. The architecture of this example system is shown in Figure 11.

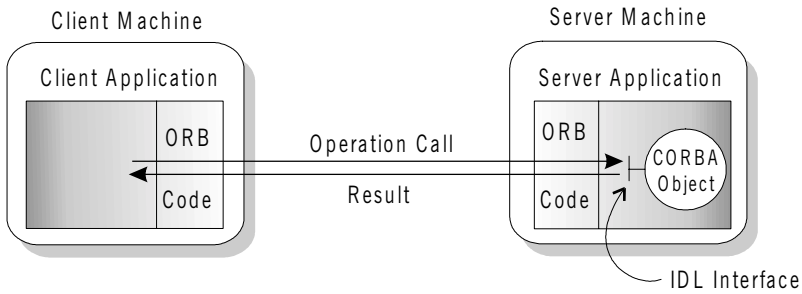


Figure 11: *Client makes a single operation call on a server*

The client and server applications communicate with each other using the Internet Inter-ORB Protocol (IIOP), which sits on top of TCP/IP. When a client invokes a remote operation, a request message is sent from the client to the server. When the operation returns, a reply message containing its return values is sent back to the client. This completes a single remote CORBA invocation.

All interaction between the client and server is mediated via a set of IDL declarations. The IDL for the Hello World! application is:

```
//IDL
interface Hello {
    string getGreeting();
};
```

The IDL declares a single `Hello` interface, which exposes a single operation `getGreeting()`. This declaration provides a language neutral interface to CORBA objects of type `Hello`.

The concrete implementation of the `Hello` CORBA object is written in Java and is provided by the server application. The server could create multiple instances of `Hello` objects if required. However, the generated code generates only one `Hello` object.

The client application has to locate the `Hello` object—it does this by reading a stringified object reference from the file `Hello.ref`. There is one operation `getGreeting()` defined on the `Hello` interface. The client invokes this operation and exits.

Development from the Command Line

Starting point code for CORBA client and server applications can be generated using the `idlgen` command line utility.

The `idlgen` utility can be used on Windows and UNIX platforms.

You implement the `Hello World!` application with the following steps:

1. Define the IDL interface, `Hello`.
2. Generate starting point code.
3. Complete the server program by implementing the single IDL `getGreeting()` operation.
4. Complete the client program by inserting a line of code to invoke the `getGreeting()` operation.
5. Build the demonstration.
6. Run the demonstration.

Define the IDL interface

Create the IDL file for the `Hello World!` application. First of all, make a directory to hold the example code:

Windows

```
> mkdir C:\OCGT\HelloExample
```

UNIX

```
% mkdir -p OCGT/HelloExample
```

Create an IDL file `C:\OCGT\HelloExample\hello.idl` (Windows) or `OCGT/HelloExample/hello.idl` (UNIX) using a text editor.

Enter the following text into the file `hello.idl`:

```
//IDL
interface Hello {
    string getGreeting();
};
```

This interface mediates the interaction between the client and the server halves of the distributed application.

Generate starting point code

Generate files for the server and client application using the CORBA Code Generation Toolkit.

In the directory `C:\OCGT\HelloExample` (Windows) or `OCGT/HelloExample` (UNIX) enter the following command:

```
idlgen java_poa_genie.tcl -all -jP HelloExample hello.idl
```

This command logs the following output to the screen while it is generating the files:

```
hello.idl:
java_poa_genie.tcl: creating idlgen/RandomFuncs.java
java_poa_genie.tcl: creating
   idlgen/HelloExample/RandomHello.java
java_poa_genie.tcl: creating idlgen/RandomHelloExample.java
java_poa_genie.tcl: creating HelloExample/HelloCaller.java
java_poa_genie.tcl: creating HelloExample/client.java
java_poa_genie.tcl: creating HelloExample/HelloImpl.java
java_poa_genie.tcl: creating HelloExample/server.java
java_poa_genie.tcl: creating build.xml
```

You can edit the following files to customize client and server applications:

Client:

`HelloExample/client.java`

Server:

`HelloExample/server.java`
`HelloExample/HelloImpl.java`

Complete the server program

Complete the implementation class, `HelloImpl`, by providing the definition of the `HelloImpl.getGreeting()` method. This Java method provides the concrete realization of the `Hello::getGreeting()` IDL operation.

Edit the `HelloImpl.java` file, and delete most of the generated boilerplate code occupying the body of the `HelloImpl.getGreeting` method Replace it with the line of code highlighted in bold font below:

```
//Java
//File 'HelloImpl.java'
...
   public java.lang.String getGreeting()
   throws org.omg.CORBA.SystemException
   {
      java.lang.String                _result;

      _result = "Hello World!";

      return _result;
   }
...
```

Complete the client program

Complete the implementation of the client `main()` function in the `client.java` file. You must add a couple of lines of code to make a remote invocation of the `getGreeting()` operation on the `Hello` object.

Edit the `client.java` file and search for the line where the `HelloExample.HelloCaller.getGreeting()` method is called. Delete this line and replace it with the line of code highlighted in bold font below:

```
//Java
//File: 'client.java'
...
    try
    {
       ...
       // Exercise interface HelloExample.Hello.
       //
       tmp_ref = read_reference("Hello.ref");
       HelloExample.Hello Hello1 =
          HelloExample.HelloHelper.narrow(tmp_ref);
       System.out.println("Greeting is: " +
   Hello1.getGreeting());

    }
    catch(Exception ex)
    {
       System.out.println("Unexpected CORBA exception: " + ex);
    }
...
```

The object reference `Hello1` refers to an instance of a `Hello` object in the server application. It is already initialized for you.

A remote invocation is made by invoking `getGreeting()` on the `Hello1` object reference. The ORB automatically establishes a network connection and sends packets across the network to invoke the `HelloImpl.getGreeting()` method in the server application.

Build the demonstration

The `itant` utility—a Java-based build tool—is used to build the generated Java code. For more details about `itant`, see http://jakarta.apache.org/ant. The `itant` utility is bundled with Orbix.

The generated file `build.xml` is used to build this demonstration. This file contains the rules for building the Hello World! application in an XML format that is understood by the `itant` utility.

To build the client and server complete the following steps:

1. Open a command line window.
2. Go to the `../OCGT/HelloExample` directory.

3. Enter:

```
> itant
```

Run the demonstration

Run the application as follows:

1. Run the Orbix services (if required).

   If you have configured Orbix to use file-based configuration, no services need to run for this demonstration. Proceed to step **2**.

   If you have configured Orbix to use configuration repository based configuration, start up the basic Orbix services.

   Open a DOS prompt in Windows, or `xterm` in UNIX. Enter:

   ```
   start_domain-name_services
   ```

   Where *`domain-name`* is the name of the configuration domain.

2. Set the Application Server Platform's environment.

   ```
   > domain-name_env
   ```

3. Run the server program.

   Open a DOS prompt, or `xterm` window (UNIX). Enter the following command:

   ```
   itant runserver
   ```

   The server outputs the following lines to the screen:

   ```
   Buildfile: build.xml

   runserver:
       [java] Initializing the ORB
       [java] Writing stringified object reference to Hello.ref
       [java] Waiting for requests...
   ```

   The server performs the following steps when it is launched:

   - It instantiates and activates a single `Hello` CORBA object.
   - The stringified object reference for the `Hello` object is written to the local `Hello.ref` file.

- The server opens an IP port and begins listening on the port for connection attempts by CORBA clients.

4. Run the client program.

   Open a new DOS prompt, or `xterm` window (UNIX). Enter the following command:

   ```
   itant runclient
   ```

   The client outputs the following lines to the screen:

   ```
   Buildfile: build.xml

   runclient:
     [java] Reading stringified object reference from Hello.ref
   Greeting is: Hello World!

   Total time: 3 seconds
   ```

   The client performs the following steps when it is run:

   - It reads the stringified object reference for the `Hello` object from the `Hello.ref` file.
   - It converts the stringified object reference into an object reference.
   - It calls the remote `Hello::getGreeting()` operation by invoking on the object reference. This causes a connection to be established with the server and the remote invocation to be performed.

5. When you are finished, terminate all processes.

   Shut down the server by typing **`Ctrl-C`** in the window where it is running.

6. Stop the Orbix services (if they are running).

   From a DOS prompt in Windows, or `xterm` in UNIX, enter:

   ```
   stop_domain-name_services
   ```

The passing of the object reference from the server to the client in this way is suitable only for simple demonstrations. Realistic server applications use the CORBA naming service to export their object references instead (see Chapter 17).

CHAPTER 3

First Application

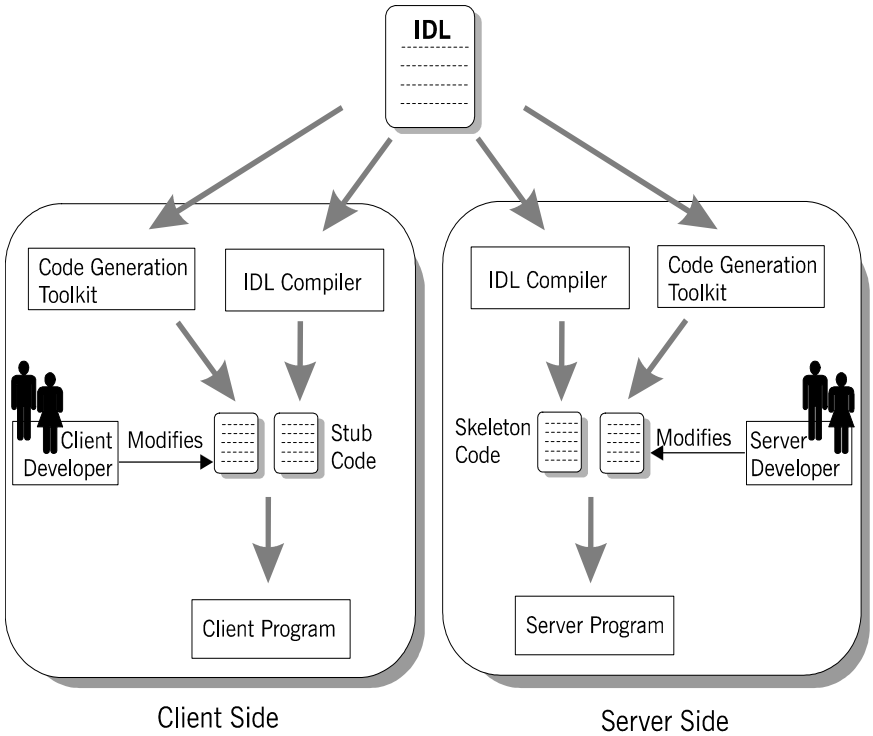
This chapter uses a simple application to describe the basic programming steps required to define CORBA objects, write server programs that implement those objects, and write client programs that access them. The programming steps are the same whether the client and server run on a single host or are distributed across a network.

In this chapter

This chapter covers the following topics:

Development Using Code Generation

With the code generation toolkit, you can automatically generate a large amount of the code required for the client and server programs:



First, you define a set of interfaces written in the OMG interface definition language (IDL). The IDL forms the basis of development for both the client and the server. The toolkit takes the IDL file as input and, based on the declarations in the IDL file, generates a complete, working Orbix application. You can then modify the generated code to add business logic to the application.

Client development

Client development consists of the following steps:

1. An IDL compiler takes the IDL file as input and generates client stub code.
2. The code generation toolkit takes the IDL file as input and generates a complete client application.

   The generated client is a dummy implementation that invokes every operation on each interface in the IDL file exactly once. The dummy client is a working application that can be built and run right away.
3. You can modify the dummy client to complete the application.

   You do not have to write boilerplate CORBA code.
4. You build the application.

   A build file is generated by the code generation toolkit.

Server development

Server development consists of the following steps:

1. An IDL compiler takes the IDL file as input and generates server skeleton code.
2. The code generation toolkit takes the IDL file as input and generates a complete server application.

   Dummy implementation classes are generated for each interface appearing in the IDL file. The dummy server is a working application that can be built and run right away.
3. You can modify the dummy server to complete the application logic.

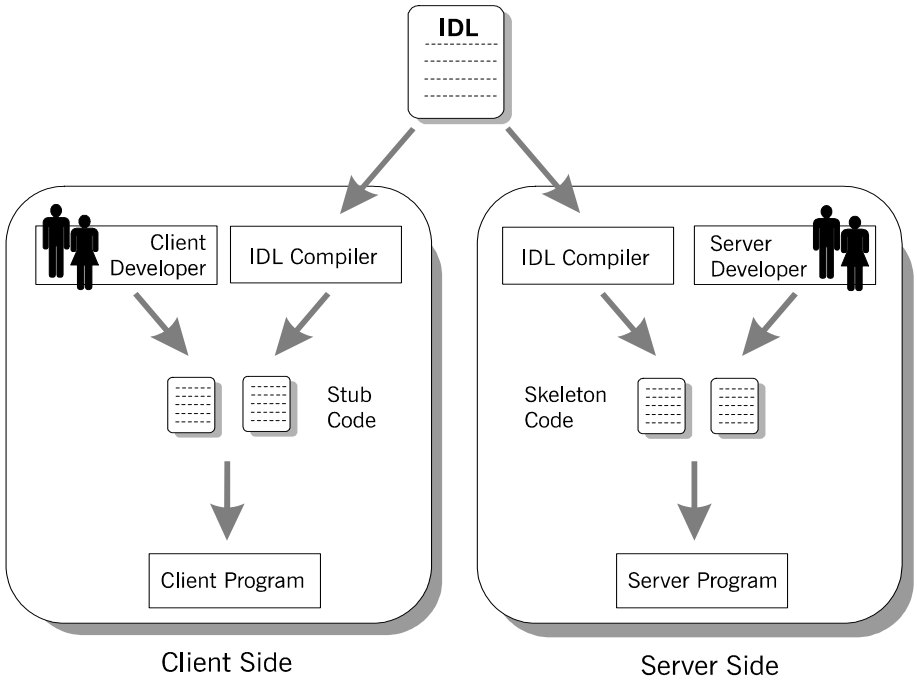
   You do not have to write boilerplate CORBA code.

   The implementations of IDL interfaces can be modified by adding business logic to the class definitions.
4. You build the application.

   A build file is generated by the code generation toolkit.

Development Without Using Code Generation

The following section outlines the steps for developing clients and servers without using the code generation toolkit (see page 40):.



First, you define a set of interfaces written in the OMG interface definition language (IDL). The IDL file forms the basis of development for both the client and the server.

Client development

Client development consists of the following steps:

1. An IDL compiler takes the IDL file as input and generates client stub code.

   The *client stub code* is a set of files that enable clients to make remote invocations on the interfaces defined in the IDL file.
2. You write the rest of the client application from scratch.

3. You build the application.

   Typically, you write a customized build file to build the client program.

Server development

Server development consists of the following steps:

1. An IDL compiler takes the IDL file as input and generates server skeleton code.

   The *server skeleton code* is a set of files that enables the server to service requests on the interfaces in the IDL file.
2. You write the rest of the server application from scratch.

   You must write an implementation class for each interface appearing in the IDL file.
3. You build the application.

   You typically write a customized build file to build the server program.

Locating CORBA Objects

Overview

Before developing an Orbix application, you must choose a strategy for locating CORBA objects.

To find a CORBA object, a client needs to know both the identity of the object and the location of the server process that provides a home for that object. In general, CORBA encapsulates both the identity and location of a CORBA object inside an entity known as an *object reference*.

In this chapter, a simple strategy is adopted to pass the object reference from the server to the client. The strategy, illustrated in Figure 12, has three steps:

1. The server converts the object reference into a string (*stringified object reference*) and writes this stringified object reference to a file.
2. The client reads the stringified object reference from the file and converts it to a real object reference.

3 The client can now make remote invocations by invoking on the object reference.

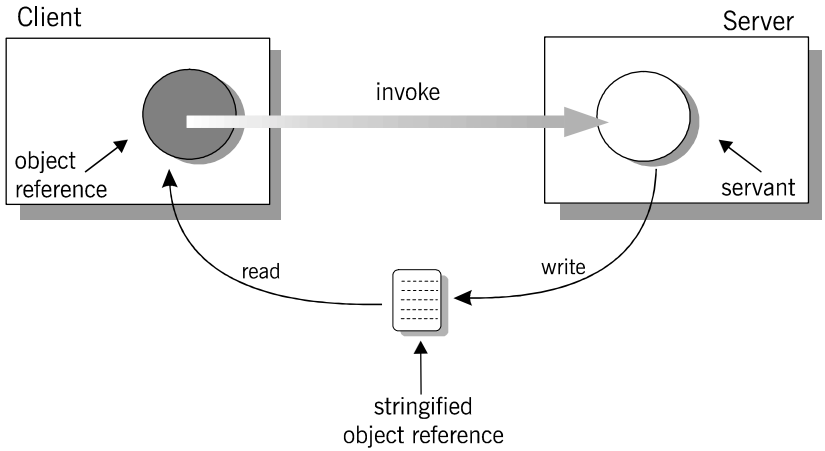


Figure 12: *Simple strategy for passing object references to clients*

This approach is convenient for simple demonstrations but is not recommended for use in realistic applications. The CORBA naming service, described in Chapter 17 on page 421, provides a more sophisticated and scalable approach to distributing object references.

Development Steps

Overview

You typically develop an Orbix application in the following steps:

1. Define IDL interfaces: Identify the objects required by the application and define their public interfaces in IDL.
2. Generate starting point code: Use the code generation toolkit to generate starting point code for the application. You can then edit the generated files to add business logic.
3. Compile the IDL definitions: The compiler generates the Java source files that you need to implement client and server programs.
4. Develop the server program: The server acts as a container for a variety of CORBA objects, each of which supports one IDL interface. You must add code to provide the business logic for each type of CORBA object.

   The server makes its CORBA objects available to clients by exporting *object references* to a well-known location.
5. Develop the client program: The client uses the IDL compiler-generated mappings to invoke operations on the object references that it obtains from the server.
6. Build the application.
7. Run the application.

Define IDL interfaces

Overview

Begin developing an Orbix enterprise application by defining the IDL interfaces to the application's objects. These interfaces implement CORBA distributed objects on a server application. They also define how clients access objects regardless of the object's location on the network.

An interface definition contains *operations* and *attributes*:

- Operations correspond to methods that clients can call on an object.
- Attributes give you access to a single data value.

  Each attribute corresponds either to a single accessor method (readonly attribute) or an accessor method and a modifier method (plain attribute).

For example, the IDL code in Example 1 defines an interface for an object that represents a building. This building object could be the beginning of a facilities management application such as a warehouse allocation system.

Example 1: *IDL for the Building Interface*

```
//IDL
//File: 'building.idl'
interface Building {
1       readonly attribute string address;

2       boolean available(in long date);
        boolean reserveDate(in long date, out long confirmation);
};
```

The IDL contains these components:

1. The `address` attribute is preceded by the IDL keyword `readonly`, so clients can read but can not set its value.
2. The `Building` interface contains two operations: `available()` and `reserveDate()`. Operation parameters can be labeled `in`, `out`, or `inout`:
   - `in` parameters are passed from the client to the object.
   - `out` parameters are passed from the object to the client.
   - `inout` parameters are passed in both directions.

`available()` lets a client test whether the building is available on a given date. This operation returns a boolean (true/false) value.

`reserveDate()` takes the date as input, returns a confirmation number as an `out` parameter, and has a boolean (true/false) return value.

All attributes and operations in an IDL interface are implicitly public. IDL interfaces have no concept of private or protected members.

Generate starting point code

Overview

It's recommended that you start developing a CORBA application by using the code generation toolkit to generate starting point code. The toolkit contains two key components:

The `idlgen` interpreter is an executable file that processes IDL files based on the instructions contained in predefined code generation scripts.

A set of *genies* (code generation scripts) are supplied with the toolkit. Most important of these is the `java_poa_genie.tcl` genie that is used to generate starting point code for a Java application.

Taking the `building.idl` IDL file as input, the `java_poa_genie.tcl` genie can produce complete source code for a distributed application that includes a client and a server program.

To generate starting point code, execute the following command:

```
idlgen java_poa_genie.tcl -all -jP BuildingExample building.idl
```

This command generates all of the files you need for this application. The `-all` flag selects a default set of genie options that are appropriate for simple demonstration applications. The `-jP` *`PackageName`* option lets you specify the name of the Java package that contains the generated code.

The main client file generated by the `java_poa_genie.tcl` genie is:

`BuildingExample/client.java`	Implementation of the client.

The main server files generated by the `java_poa_genie.tcl` genie are:

`BuildingExample/server.java`	Server `main()` containing the server initialization code.
`BuildingExample/BuildingImpl.java`	Implementation of the `BuildingImpl` servant class.

One file is generated for building the application: build.xml, which is an XML file that contains the rules for building the Hello World! application.

The files in the generated `idlgen` directory are used to support a dummy implementation of the client and server programs:

Dummy implementation of client and server programs

The generated starting point code provides a complete dummy implementation of the client and the server programs. The dummy implementation provides:

- A server program that implements every IDL interface.

  Every IDL operation is implemented with default code. Return values, `inout` and `out` parameters are populated with randomly generated values. At random intervals a CORBA user exception might be thrown instead.
- A client program that calls every operation on every IDL interface once.

The dummy client and server programs can be built and run as they are.

Modifying dummy client and server programs

Later steps describe in detail how to modify the generated code to implement the business logic of the `Building` application.

In the code listings that follow, modifications are indicated as follows:

- Additions to the generated code are highlighted in bold font. You can manually add this code to the generated files using a text editor.
- In some cases the highlighted additions replace existing generated code, requiring you to manually delete the old code.

Compile the IDL definitions

Overview

This step is optional if you use the code generation toolkit to develop an application. The build.xml file generated by the toolkit has a rule to run the IDL compiler automatically.

After defining your IDL, compile it using the CORBA IDL compiler. The IDL compiler checks the validity of the specification and generates code in Java that you use to write the client and server programs.

Compile the `Building` interface by running the IDL compiler as follows:

```
idl -jbase=-PBuildingExample:-Ojava_output
   -jpoa=-PBuildingExample:-Ojava_output building.idl
```

The `-jbase` option generates Java client stub code. The `-PBuildingExample` sub-option puts the stub code in the `BuildingExample` Java package. The `-Ojava_output` sub-option puts stub code files in the `java_output` directory.

The `-jpoa` option generates server-side code for the POA in Java. The `-jpoa` sub-options are analogous to the `-jbase` sub-options.

Run the IDL compiler with the `-flags` option to get a complete description of the supported options.

Output from IDL compilation

The IDL compiler produces several Java files when it compiles the `building.idl` file. These files contain Java definitions that correspond to your IDL definitions. You should never modify this code.

The generated files can be divided into two categories:

- Client stub code is compiled and linked with client programs, so they can make remote invocations on `Building` CORBA objects.
- Server skeleton code is compiled and linked with server programs, so they can service invocations on `Building` CORBA objects.

Client stub code

The stub code is used by clients and consists of the following files:

`Building.java`	A file defining a Java `Building` interface. Clients use this Java interface to invoke IDL `Building` operations.

`BuildingHelper.java`	A file defining a Java `BuildingHelper` class. Every user-defined IDL type has an associated Java `Helper` class.
`BuildingHolder.java`	A file defining a Java `BuildingHolder` class. Clients use this class to pass `inout` and `out` parameters. Every IDL type has an associated Java `Holder` class.
`_BuildingStub.java`	A file containing stub code that enables remote access to `Building` objects—not directly used by clients.
`BuildingOperations.java`	A file containing the Java `BuildingOperations` interface—not directly used by clients.

Server skeleton code

The skeleton code is a superset of the stub code. The additional files contain code that allows you to implement servants for the `Building` interface. The skeleton code consists of the following files:

`BuildingPOA.java`	A file containing the `BuildingPOA` class. Servers can use this class to implement the IDL `Building` interface.
`BuildingPOATie.java`	A file containing the `BuildingPOATie` class. This class provides an alternative approach to implementing the IDL `Building` interface, known as *the tie approach*.

IDL to Java mapping

The IDL compiler translates IDL into stub and skeleton code for a given language—in this case, Java. As long as the client and server programs comply with the definitions in the generated stub and skeleton code, the runtime ORB enables type-safe interaction between the client and the server.

Example 2: *Java Stub Code for the Building Interface*

```
// File: 'Building.java'
package BuildingExample;
```

Example 2: *Java Stub Code for the Building Interface*

```
1   public interface Building
        extends BuildingOperations,
        org.omg.CORBA.object,
        org.omg.CORBA.portable.IDLEntity
    {
    }
    ...
    // File: 'BuildingOperations.java'
    package BuildingExample;

    public interface BuildingOperations
    {
2       java.lang.String address();

3       boolean available(int date);

4       boolean reserveDate(
            int date,
            org.omg.CORBA.IntHolder confirmation
        );
    }
```

The code can be explained as follows:

1. The Java `Building` interface provides the client view of a CORBA object. The methods inherited from the Java `BuildingOperations` interface correspond to the attributes and operations of the IDL `Building` interface.

   When a client program calls methods on an object of `Building` type, Orbix forwards the method calls to a server object that supports the IDL `Building` interface.

2. The Java `address()` method is mapped from the IDL readonly `address` attribute. Clients call this method to get the attribute's current value, which returns a `java.lang.String`.

3. The Java `available()` method is mapped from the IDL `available()` operation. The parameter and return type are mapped as follows:

   - The `date` parameter (`in` parameter) is mapped from an IDL `long` to a Java `int`.
   - The return type is mapped from an IDL `boolean` to a Java `boolean`.

4. The Java `reserveDate()` method is mapped from the IDL `reserveDate()` operation. The parameters and return type are mapped as follows:
   - The `date` parameter (`in` parameter) is mapped from an IDL `long` to a Java `int`.
   - The `confirmation` parameter (`out` parameter) is mapped from an IDL `long` to a Java `org.omg.CORBA.IntHolder` object.
   - The return type is mapped from an IDL `boolean` to a Java `boolean`.

   All `inout` and `out` parameters are declared as `Holder` types in Java. The `org.omg.CORBA.IntHolder` type is used to pass the `confirmation` parameter from the server back to the client. For an example of how to use the `IntHolder` type, see "Client business logic" on page 63.

Develop the server program

The main programming task on the server side is the implementation of servant classes. In this demonstration there is one interface, `Building`, and one corresponding servant class, `BuildingImpl`. The code generation toolkit generates a dummy definition of every servant class. The `BuildingImpl` servant class is defined in the `BuildingImpl.java` file.

The other programming task on the server side is the implementation of the server `main()`. For this simple demonstration, the generated server `main()` does not require any modification. It is discussed in detail in "Enhancing Server Functionality" on page 69.

Define the servant class

The code generation toolkit generates the `BuildingImpl.java` file, which contains an outline of the method definitions for the `BuildingImpl` servant class. You should edit this file to fill in the bodies of methods that correspond to the operations and attributes of the `Building` interface. It is usually necessary to edit the constructor of the servant class as well.

Manual additions made to the generated code are shown in bold font. In some cases, the additions replace existing generated code requiring you to manually delete the old code.

Example 3: *Java BuildingImpl Servant Implementation*

```
// File: 'BuildingImpl.java'
...
package BuildingExample;

// CORBA imports
import org.omg.CORBA.ORB;
import org.omg.CORBA.StringHolder;

1  public class BuildingImpl extends BuildingPOA
   {
```

Example 3: *Java BuildingImpl Servant Implementation*

```
2   //----------------------
    // Private Member Variables
    //----------------------
    private int        m_confirmation_counter;
    private int[]      m_reservation;

    boolean isClient = false;
    org.omg.PortableServer.POA m_poa = null;
    /**
     * The state for the CORBA Attribute 'address'
     */
    protected       java.lang.String               m_address;

3   public static  BuildingImpl
     _create(org.omg.PortableServer.POA the_poa)
      throws org.omg.CORBA.SystemException
    {
      return new BuildingImpl(the_poa);
    }

4   public BuildingImpl(
      org.omg.PortableServer.POA the_poa
    )
    {
      m_address = "200 West Street, Waltham, MA.";
      m_confirmation_counter = 1;
      m_reservation = new int[366];
      for (int i=0; i<366; i++) { m_reservation[i] = 0; }

      m_poa = the_poa;

      System.out.println("created");
    }


    /**
     * implementation for IDL operation available().
     */
    public
     boolean available(
              int                             date
     )
      throws org.omg.CORBA.SystemException
    {
```

Example 3: *Java BuildingImpl Servant Implementation*

```
5       if (1<=date && date<=366) {
            return (m_reservation[date-1]==0);
        }
        return true;
    }

    /**
     * implementation for IDL operation reserveDate().
     */
    public
     boolean reserveDate(
              int                           date,
              org.omg.CORBA.IntHolder       confirmation
     )
      throws org.omg.CORBA.SystemException
    {
6       confirmation.value = 0;

        if (1<=date && date<=366) {
            if (m_reservation[date-1]==0) {
                m_reservation[date-1] = m_confirmation_counter;
                confirmation.value = m_confirmation_counter;
                m_confirmation_counter++;
                return true;
            }
        }
        return false;
    }

    /**
     * Implementation for IDL address accessor.
     */
    public
     java.lang.String address()
    {
7       return m_address;
    }

8   public org.omg.PortableServer.POA _default_POA()
    {
       return m_poa;
    }
}
```

The code can be explained as follows:

1. The `BuildingImpl` servant class inherits from `BuildingPOA`.

   The `BuildingPOA` class is a standard name for the base class generated for the `Building` interface. By inheriting from `BuildingPOA`, you are indicating to the ORB that `BuildingImpl` is the servant class that implements the `Building` interface. This approach to associating a servant class with an interface is called the *inheritance approach*.

2. The lines of code shown in bold font are added to the generated code to complete the application. Two private member variables are declared to store the state of a `BuildingImpl` object.

   - The `m_confirmation_counter` index counter is incremented each time a reservation is confirmed.
   - The `m_reservation` array has 366 elements (representing the 365 or 366 days in a year). The elements are equal to zero when unreserved or a positive integer (the confirmation number) when reserved.

3. `_create()` is a `BuildingImpl` method that creates `BuildingImpl` instances.

   Note: `_create()` is not a standard part of CORBA. It is generated by the code generation toolkit for convenience. You are free to call the constructor directly, or remove the `_create()` method entirely if you wish.

4. The `BuildingImpl` constructor is an appropriate place to initialize any member variables. The three private member variables—`m_address`, `m_confirmation_counter` and `m_reservation`—are initialized here. Replace the dummy initialization code with the highlighted code.

5. The few lines of code here implement `available()` and replace the generated dummy code. If an element of the array `m_reservation` is zero, that means the date is available. Otherwise the array element holds the confirmation number (a positive integer).

6. The few lines of code here implement `reserveDate()` and replace the generated dummy code. Because `confirmation` is declared as an `out` parameter in IDL, it is passed as an `org.omg.CORBA.IntHolder` type. The value of the `confirmation` variable is accessed as `confirmation.value`.

   The use of holder types gets around the Java language feature that limits parameter passing to pass-by-value. Changes made to `confirmation.value` can be seen by the calling code. Effectively, the holder types allow you to imitate pass-by-reference in Java.

7. The `address()` accessor method is implemented by returning a reference to the `m_address` string.

8. `_default_POA()` is inherited from `org.omg.PortableServer.Servant` by way of `BuildingPOA`. It is a standard servant method that identifies the POA object with which this servant is associated. In this example, the value of `m_poa` is set in the `BuildingImpl` constructor.

   `_default_POA()` is overridden to guard against the possibility of accidental implicit activation. For information about implicit activation, see page 214.

Develop the client program

Overview

The generated code in the `client.java` file takes care of initializing the ORB and getting a `Building` object reference. This allows the client programmer to focus on the business logic of the client application.

You modify the generated client code by implementing the logic of the client program. Use the `Bulding` object reference to access an object's attributes and invoke its operations.

Client main()

The code in the client `main()` initializes the ORB, reads a `Building` object reference from the file `Building.ref` and hands over control to `run_warehouse_menu()`, which is described in the next section. When `run_warehouse_menu()` returns, the generated code shuts down the ORB.

Changes or additions to the code are shown in bold font.

Example 4: *Java Client main() Funciton*

```
//File: 'client.java'
...
package BuildingExample;

import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import java.io.*;

public class client
{

  // global_orb -- make ORB global so all code can find it.
1   public static org.omg.CORBA.ORB global_orb = null;
```

Example 4: *Java Client main() Funciton*

```
2   static org.omg.CORBA.Object read_reference(String file)
    {
      System.out.println(
          "Reading stringified object reference from " + file
      );
      String ref = null;
      try {
        FileReader retrieve=new FileReader(file);
        BufferedReader in=new BufferedReader(retrieve);
        ref = in.readLine();
      }
      catch (IOException ex) {
        System.out.println(
            "Error reading object reference from "
            + file + " : " + ex.toString()
        );
        return null;
      }
     org.omg.CORBA.Object obj = global_orb.string_to_object(ref);
      return obj;
    }
    ...

    // main() -- the main client program.
    public static void main (String args[])
    {
      try {
          // For temporary object references
          org.omg.CORBA.Object tmp_ref;

          // Initialise the ORB
3         global_orb = ORB.init(args, null);

          // Exercise the BuildingExample.Building interface.
4         tmp_ref = read_reference("Building.ref");
5         BuildingExample.Building Building1 =
              BuildingExample.BuildingHelper.narrow(tmp_ref);
6         run_warehouse_menu(Building1);
      }
      catch(Exception ex) {
          System.out.println("Unexpected CORBA exception: " + ex);
      }

      // Ensure that the ORB is properly shutdown and cleaned up
      try  {
```

Example 4: *Java Client main() Funciton*

```
7           global_orb.shutdown(true);
        }
        catch (Exception ex) {
            // Do nothing.
        }
        return;
    }
}
```

The code can be explained as follows:

1. Declare the variable `global_orb` in the global scope so that all parts of the program can easily access the ORB object.

   The `global_orb` is temporarily set equal to `null`, which represents a *nil object reference*.

2. Define `read_reference()` to read an object reference from a file. This method reads a stringified object reference from a file and converts the stringified object reference to an object reference using `org.omg.CORBA.ORB.string_to_object()`. The return type of `read_reference()` is `org.omg.CORBA.Object`—the base type for object references.

   If there is an error, `read_reference()` returns `null`.

3. Call `org.omg.CORBA.ORB.init()` to get an object reference to an ORB object.

   A client must associate itself with an ORB in order to get object references to CORBA services such as the naming service or trader service.

4. Get a reference to a `Building` CORBA object by calling `read_reference()`, passing the name of the file, `Building.ref`, that contains its stringified object reference.

5. Narrow the CORBA object to a `Building` object, to get the object reference, `Building1`.

   Every IDL interface has an associated `Helper` class in Java. For example, the `Building` interface has a `BuildingExample.BuildingHelper` class. The `Helper` class defines a static `narrow()` method to let you narrow an object reference from a

base type to a derived type. It is similar to a Java cast operation, but is used specifically for types related via IDL inheritance.

6. Replace the lines of generated code that use the `BuildingCaller` class with a single call to `run_warehouse_menu()`.

   `run_warehouse_menu()` uses the `Building1` object reference to make remote invocations on the server.

7. The ORB must be explicitly shut down before the client exits.

   `CORBA::ORB::shutdown()` stops all server processing, deactivates all POA managers, destroys all POAs, and causes the `run()` loop to terminate. The boolean argument, `true`, indicates that `shutdown()` blocks until shutdown is complete.

When an object reference enters a client's address space, Orbix creates a *proxy object* that acts as a stand-in for the remote servant object. Orbix forwards method calls on the proxy object to corresponding servant object methods.

Client business logic

You access an object's attributes and operations by calling the appropriate `Building` class methods on the proxy object. The proxy object redirects the Java calls across the network to the appropriate servant method.

The following code uses the Java member access operator (`.`) on the `Building` object `warehouse` to access `Building` class methods.

Additions to the code are shown in bold font.

Example 5: *Java Client Business Logic*

```
//File: 'client.java'
import org.omg.CORBA.*;
...
public class client
{
  ...
  public static void run_warehouse_menu(Building warehouse)
  {
    String address = warehouse.address();
    System.out.println("The warehouse address is:\n" + address);

    InputStreamReader userInputStream
        = new InputStreamReader(System.in);
    BufferedReader userBuf = new BufferedReader(userInputStream);

    int date;
1   IntHolder confirmationH = new IntHolder();
    String quit = "n";
    try {
        do {
            System.out.println(
                "Enter day to reserve warehouse (1,2,...): "
            );
            date = Integer.parseInt(userBuf.readLine());
            if(warehouse.available(date)) {
2              if (warehouse.reserveDate(date, confirmationH) ) {
                    System.out.println(
```

Example 5: *Java Client Business Logic*

```
3                       "Confirmation number: " +
    confirmationH.value
                    );
                }
                else {
                    System.out.println(
                        "Reservation attempt failed!"
                    );
                }
            }
            else {
                System.out.println("That date is unavailable.");
            }
            System.out.println("Quit? (y,n)");
            quit = userBuf.readLine();
        }
        while (quit.equals("n"));
    }
    catch (java.io.IOException ex) {
        System.err.println("error: failed to read user input");
    }
  }
  ...
};
```

The `org.omg.CORBA.IntHolder` type is used as follows:

1. Because `confirmation` is an `out` parameter, a holder type (of `org.omg.CORBA.IntHolder` type) must be allocated for it.
2. The content of the `confirmationH` holder type, `confirmationH.value`, does not need to be initialized before the operation invocation.

   After invoking `reserveDate()`, `confirmationH.value` holds the returned `out` parameter value.
3. The confirmation number is accessed as `confirmationH.value`.

Build the application

The tool used to build the generated Java code is the `itant` utility, which is a Java-based build tool developed by Apache as part of the Jakarta project. For further details about `itant`, see http://jakarta.apache.org/ant. The `itant` utility is bundled with Orbix.

The file `build.xml` is generated when building this demonstration. This file sets up your environment to use the `itant` utility. This file contains the rules for building the Hello World! application in an XML format that is understood by the `itant` utility.

To build the client and server, go to the example directory and at a command line prompt enter:

```
> itant build_all
```

Run the application

Prerequisites

The prerequisites for running this application are:

- The Orbix deployment environment is installed on the machine where the demonstration is run.
- Orbix has been correctly configured. See the *Application Server Platform Administrator's Guide* for details.
- Your Java development kit (JDK) is configured to use the Orbix ORB runtime (see "Setting ORB Properties for the Orbix ORB" on page 28).

This demonstration assumes that both the client and the server run in the same directory.

Steps

Perform the following steps to run the application:

1 Run the Orbix services (if required).

If you have configured Orbix to use file-based configuration, no services need to run for this demonstration. Proceed to step **2**.

If you have configured Orbix to use configuration repository based configuration, start up the basic Orbix services.

Open a new DOS prompt in Windows, or `xterm` in UNIX. Enter:

`start_`*`domain-name`*`_services`

where *`domain-name`* is the name of the default configuration domain.

2 Run the server program.

Open a new DOS prompt in Windows, or `xterm` in UNIX. Enter the following commands:

```
set CLASSPATH=%CLASSPATH%;./classes
java BuildingExample.server
```

The server outputs the following lines to the screen:

```
Initializing the ORB
Writing stringified object reference to Building.ref
Waiting for requests...
```

At this point the server is blocked while executing `ORB.run()`.

3 Run the client program.

Open a new DOS prompt in Windows, or `xterm` in UNIX. Enter the following command:

```
set CLASSPATH=%CLASSPATH%;./classes
java BuildingExample.client
```

4 When you are finished, terminate all processes.

The server can be shut down by typing Ctrl-C in the window where it is running.

5 Stop the Orbix services (if they are running).

From a DOS prompt in Windows, or `xterm` in UNIX, enter:

`stop_`*`domain-name`*`_services`

where *`domain-name`* is the name of the default configuration domain.

Enhancing Server Functionality

Overview

In this demonstration, the default implementation of `main()` suffices so there is no need to edit the `server.java` file.

However, for realistic applications, you need to customize the server `main()` to specify what kind of POAs are created. You also need to select which CORBA objects get activated as the server boots up.

The default server `main()` contains code to perform these tasks:

1. Initialize the ORB.
2. Create a POA for transient objects.
3. Create servant objects.
4. Activate CORBA objects—the default server code activates one CORBA object for each of the interfaces defined in the IDL file.
5. Export object references—an object reference is exported for each of the activated CORBA objects.
6. Activate the POA manager—so it can process requests on the CORBA objects it manages.
7. Shut down the ORB—shut down the ORB cleanly before exiting.

In this demonstration, there is only one interface, `Building`, and a single CORBA object of this type is activated.

The following subsections discuss the code in the `server.java` file piece by piece. For a complete source listing of `server.java`, see page 81.

Initialize the ORB

Before a server can make its objects available to the rest of an enterprise application, it must initialize the ORB:

Example 6: *Java Initializing the ORB*

```
...
public class server {
1   public static ORB global_orb = null;
  ...
  public static void main(String args[])
  {
    ...
    try {
2       global_orb = ORB.init(args, null);
```

The code can be explained as follows:

1. The `global_orb` variable is used to hold a reference to an `org.omg.CORBA.ORB` object.
2. `org.omg.CORBA.ORB.init()` is used to create an instance of an ORB. Command-line arguments are passed to the ORB via the `args` parameter. `ORB.init()` searches `args` for arguments of the general form `-ORB`*suffix*, parses these arguments, and removes them from the argument list.

The second parameter (properties parameter) of `ORB.init()` is usually left equal to `null`. The following sub-subsection describes how the properties parameter can (optionally) be used to set the `org.omg.CORBA.ORBClass` property.

Programatically setting the orbclass property

This step is *not* recommended for most Java Orbix applications. See "Setting ORB Properties for the Orbix ORB" on page 28 for the recommended ways of setting the ORB properties.

The `org.omg.CORBA.ORBClass` property can be set programatically using the `Properties` parameter of `ORB.init()`, as in the following example:

Example 7:

```
public static ORB global_orb = null;
    ...
    public static void main(String args[])
    {
1       java.util.Properties p = new java.util.Properties();
        p.setProperty("org.omg.CORBA.ORBClass",
                      "com.iona.corba.art.artimpl.ORBImpl");
        ...
        try {
2           global_orb = ORB.init(args, p);
            ...
```

The code can be explained as follows:

1. A `java.util.Properties` object is created that can hold one or more property values. The `org.omg.CORBA.ORBClass` property is set on the `p` property object.
2. The property object is passed as the second argument to the `ORB.init()` call, which returns a new ORB object that is implemented by the `com.iona.corba.art.artimpl.ORBImpl` class.

Note: The `org.omg.CORBA.ORBSingletonClass` property *cannot* be set programatically because it is used in a different context to `ORB.init(args,p)`— that is, the no argument `ORB.init()` call. The `org.omg.CORBA.ORBSingletonClass` property is searched for in a static initializer on `org.omg.CORBA.ORB`, and can only be usefully set in the system properties or the `iona.properties` file.

Create a POA for transient objects

A simple POA object is created using the following lines of code:

Example 8:

```
...
    try {
        ...
1       tmp_ref
            = global_orb.resolve_initial_references("RootPOA");
    }
    catch (org.omg.CORBA.ORBPackage.InvalidName ex) {
        // Handle exception...
    }
2   POA root_poa = POAHelper.narrow(tmp_ref);
3   POAManager root_poa_manager = root_poa.the_POAManager();

4   // Now create our own POA.
    POA my_poa = create_simple_poa("my_poa",
                                   root_poa,
                                   root_poa_manager);
```

The code can be explained as follows:

1. Get a reference to the root POA object by calling `resolve_initial_references("RootPOA")` on the ORB.

   `resolve_initial_references()` provides a bootstrap mechanism for obtaining access to key Orbix objects. It contains a mapping of well-known names to important objects such as the root POA (`RootPOA`), the naming service (`NameService`), and other objects and services.

2. Narrow the root POA reference `tmp_ref` to type `PortableServer.POA` using `PortableServer.POAHelper.narrow()`.

   Because `tmp_ref` is of `org.omg.CORBA.Object` type—the generic base class for object references—methods specific to the `PortableServer.POA` class are not directly accessible. It is therefore necessary to down-cast the `tmp_ref` pointer to the actual type of the object reference using `POAHelper.narrow()`.

3. Obtain a reference to the root POA manager object.

A POA manager controls the flow of messages to a set of POAs. CORBA invocations cannot be processed unless the POA manager is in an active state (see page 79).

4. Create the `my_poa` POA as a child of `root_poa`. The `my_poa` POA becomes associated with the `root_poa_manager` POA manager. This means that the `root_poa_manager` object controls the flow of messages into `my_poa`.

---

create_simple_poa()

The `create_simple_poa()` method is defined as follows:

```
...
  static POA create_simple_poa(
    String                    poa_name,
    POA                       parent_poa,
    POAManager                poa_manager
  )
  {
    // Create a policy list.
    // Policies not set in the list get default values.
    org.omg.CORBA.Policy[] policies = new
   org.omg.CORBA.Policy[1];
    int i = 0;
    POA new_poa = null;

    // Make the POA single threaded.
    policies[i++] = parent_poa.create_thread_policy(
        ThreadPolicyValue.SINGLE_THREAD_MODEL
    );
    if(i>1 || i<1) {
      System.out.println("Policy creation failed");
      System.exit(1);
    }
```

```
    try {
      new_poa = parent_poa.create_POA(poa_name,
                                      poa_manager,
                                      policies);
    }
    catch (
       org.omg.PortableServer.POAPackage.AdapterAlreadyExists ex
      ) {
      System.out.println(
        "Failed to create POA with exception: " +ex.toString() );
      System.exit(1);
    }
    catch (org.omg.PortableServer.POAPackage.InvalidPolicy ex) {
      System.out.println(
        "Failed to create POA with exception: " +ex.toString() );
      System.exit(1);
    }
    return new_poa;
  }
```

A POA is created by invoking `PortableServer.POA.create_POA()` on an existing POA object. The POA on which this method is invoked is known as the *parent POA* and the newly created POA is known as the *child POA*.

`create_POA()` takes the following arguments:

- `poa_name` is the adapter name. This name is used within the ORB to identify the POA instance relative to its parent.
- `poa_manager` is a reference to a POA manager object with which the newly created POA becomes associated.
- `policies` is a list of policies that configure the new POA. For more information, see "Using POA Policies" on page 202.

The POA instance returned by `create_simple_poa()` accepts default values for most of its policies. The resulting POA is suitable for activating *transient CORBA objects*. A transient CORBA object is an object that exists only as long as the server process that created it. When the server is restarted, old transient objects are no longer accessible.

Create servant objects

Overview

A number of servant objects must be created. A servant is an object that does the work for a CORBA object. For example, the `BuildingImpl` servant class contains the code that implements the `Building` IDL interface.

A single `BuildingImpl` servant object is created as follows:

```
...
// Variables to hold our servants
Servant the_Building = null;
...
// Create a servant for BuildingExample.Building.
the_Building = BuildingExample.BuildingImpl._create(my_poa);
```

In this example, `_create()`creates an instance of a `BuildingImpl` servant. The POA reference `my_poa` that is passed to `_create()` must be the same POA that is used to activate the object in the next section "Activate CORBA objects".

`_create()`is not a standard CORBA method. It is a convenient pattern implemented by the code generation toolkit. You can use the `BuildingImpl` constructor instead, if you prefer.

Activate CORBA objects

Overview

A CORBA object must be activated before it can accept client invocations. Activation is the step that establishes the link between an ORB, which receives invocations from clients, and a servant object, which processes these invocations.

In this step, two fundamental entities are created that are closely associated with a CORBA object:

- An object ID.

  This is a CORBA object identifier that is unique with respect to a particular POA instance. In the case of a persistent CORBA object, the object ID is often a database key that is used to retrieve the state of the CORBA object from the database.
- An object reference.

  This is a handle on a CORBA object that exposes a set of methods mapped from the operations of its corresponding IDL interface. It can be stringified and exported to client programs. Once a client gets hold of an object reference, the client can use it to make remote invocations on the CORBA object.

A single `Building` object is activated using the following code:

Example 9:

```
org.omg.CORBA.Object tmp_ref = null;
...
byte [] oid;
...
1 oid = my_poa.activate_object(the_Building);
2 tmp_ref = my_poa.id_to_reference(oid);
```

The code can be explained as follows:

1. Activate the CORBA object.

   A number of things happen when `activate_object()` is called:

   - An unique object ID, `oid`, is automatically generated by `my_poa` to represent the CORBA object's identity. Automatically generated object IDs are convenient for use with transient objects.

- The CORBA object becomes associated with the POA, `my_poa`.
- The POA records the fact that the `the_Building` servant provides the implementation for the CORBA object identified by `oid`.

2. Use `org.omg.PortableServer.POA.id_to_reference()` to generate an object reference, `tmp_ref`, from the given object ID.

You can activate a CORBA object in various ways, depending on the policies used to create the POA. For information about activating objects in the POA, see "Activating CORBA Objects" on page 185; for information about activating objects on demand, see Chapter 11 on page 271.

Export object references

Overview

A server must advertise its objects so that clients can find them. In this demonstration, the `Building` object reference is exported to clients using `write_reference()`:

```
write_reference(tmp_ref,"Building.ref");
```

This call writes the `tmp_ref` object reference to the `Building.ref` file.

`write_reference()` writes an object reference to a file in stringified form. It is defined as follows:

```
static void write_reference(
    org.omg.CORBA.Object            ref,
    String                          objref_file
)
{
    String stringified_ref = global_orb.object_to_string(ref);
    System.out.println(
        "Writing stringified object reference to " + objref_file
    );

    try {
      FileWriter store = new FileWriter(objref_file);
      store.write(stringified_ref);
      store.flush();
      store.close();
    }
    catch (IOException ex) {
      System.out.println("Failed to write to " + objref_file);
    }
}
```

The `ref` object reference is converted to a string by passing `ref` as an argument to `org.omg.CORBA.ORB.object_to_string()`. The string is then written to the `objref_file` file.

CORBA clients can read the `objref_file` file to obtain the object reference.

This approach to exporting object references is convenient to use for this simple demonstration. Realistic applications, however, are more likely to use the CORBA naming service instead.

Activate the POA manager

Overview

After a server has set up the objects and associations it requires during initialization, it must tell the ORB to start listening for requests:

Example 10:

```
1  // Activate the POA Manager.
   //
   try{
       root_poa_manager.activate();
   }
   catch (
       org.omg.PortableServer.POAManagerPackage.AdapterInactive
      ex){
       // Handle exception...
   }
2  global_orb.run();
```

The code can be explained as follows:

1. A POA manager acts as a gatekeeper for incoming object requests. The manager can be in four different states: *active*, *holding*, *discarding*, or *inactive* (see Table 10 on page 218). A POA manager can accept object requests only after it is activated by calling `org.omg.PortableServer.POAManager.activate()`.
2. `org.omg.CORBA.ORB.run()` puts the ORB into a state where it listens for client connection attempts and accepts request messages from existing client connections.

   `org.omg.CORBA.ORB.run()` is a blocking method that returns only when `org.omg.CORBA.ORB.shutdown()` is invoked.

Shut down the ORB

Overview

Shutdown is initiated when a Ctrl-C or similar event is sent to the server from any source. You can shut down the server application as follows:

- On Windows platforms, switch focus to the MS-DOS box where the server is running and type Ctrl-C.
- On UNIX platforms, switch focus to the xterm window where the server is running and type Ctrl-C.
- On UNIX, send a signal to a background server process using the `kill` system command.

With JDK 1.2, there is no mechanism for the Java Virtual Machine to detect abnormal program termination (for example, Ctrl-C to exit). It is, therefore, unlikely that `orb.shutdown()` is ever called but it is good programming practice to call it before exit, as in the current server example.

With JDK 1.3, an API for Java Virtual Machine shutdown hooks has been added to the `java.lang.Runtime` class that provides process-shutdown notification. A JDK 1.3 application can initiate shutdown actions, such as `orb.shutdown()`, before the Java Virtual Machine exits.

See the release notes for the JDK 1.3 in the documentation pages at SUN's website, `http://java.sun.com`, for further details.

Complete Source Code for server.java

```
//Java
//------------------------------------------------------------------
   -
// Edit idlgen config file to get your own copyright notice
// placed here.
//------------------------------------------------------------------
   -

// Automatically generated server for the following
// IDL interfaces:
//   Building

package BuildingExample;

import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import java.io.*;

import java.text.DateFormat;

/**
 * server:  This class implements the CORBA server automatically
 * generated from the idl file.
 *
 */

public class server {

  public static ORB global_orb = null;

  // write_reference() -- export object reference to file.
  // This is a useful way to advertise objects for simple tests
  // and demos.
  // The CORBA naming service is a more scalable way to
  // advertise references.
  //
  static void write_reference(
    org.omg.CORBA.Object            ref,
    String                          objref_file
  )
```

```
{
  String stringified_ref = global_orb.object_to_string(ref);
  System.out.println(
      "Writing stringified object reference to " + objref_file
  );

  try {
    FileWriter store = new FileWriter(objref_file);
    store.write(stringified_ref);
    store.flush();
    store.close();
  }
  catch (IOException ex) {
    System.out.println("Failed to write to " + objref_file);
  }
}

// create_simple_poa() --
// Create a POA for simple servant management.
static POA create_simple_poa(
  String                    poa_name,
  POA                       parent_poa,
  POAManager                poa_manager
)

{
  // Create a policy list.
  // Policies not set in the list get default values.
  org.omg.CORBA.Policy[] policies = new
 org.omg.CORBA.Policy[1];
  int i = 0;
  POA new_poa = null;
  // Make the POA single threaded.
  //
  policies[i++] = parent_poa.create_thread_policy(
      ThreadPolicyValue.SINGLE_THREAD_MODEL
  );
  if(i>1 || i<1)
  {
    System.out.println("Policy creation failed");
    System.exit(1);
  }
```

```
  try {
    new_poa = parent_poa.create_POA(poa_name,
                                    poa_manager,
                                    policies);
  }
  catch (
      org.omg.PortableServer.POAPackage.AdapterAlreadyExists ex
  )
  {
    System.out.println(
      "Failed to create POA with exception: " +ex.toString()
    );
    System.exit(1);
  }
  catch (org.omg.PortableServer.POAPackage.InvalidPolicy ex)
  {
    System.out.println(
      "Failed to create the POA with exception : "
 +ex.toString()
    );
    System.exit(1);
  }
  return new_poa;
}

// main() -- set up a POA, create and export object references
public static void main(String args[])
{
  // Variables to hold our servants
  Servant the_Building = null;

  try {
      // For temporary object references
      org.omg.CORBA.Object tmp_ref = null;
```

```
        // Initialise the ORB and Root POA.
        //
        System.out.println("Initializing the ORB");
        try {
          global_orb = ORB.init(args, null);
          tmp_ref
              = global_orb.resolve_initial_references("RootPOA");
        }
        catch (org.omg.CORBA.ORBPackage.InvalidName ex) {
          System.out.println(
              "Caught exception while resolving to  RootPOA : "
              + ex.toString()
          );
          System.exit(1);
        }

        POA root_poa = POAHelper.narrow(tmp_ref);
        POAManager root_poa_manager = root_poa.the_POAManager();

        // Now create our own POA
        POA my_poa = create_simple_poa("my_poa",
                                       root_poa,
                                       root_poa_manager);

        // Create servants and export object references
        // Note: _create is a useful convenience function created
        // by the genie; it is not a standard CORBA function
        byte [] oid;
```

```
        try{
          // Create a servant for BuildingExample.Building
          the_Building =
              BuildingExample.BuildingImpl._create(my_poa);
          oid = my_poa.activate_object(the_Building);
          tmp_ref = my_poa.id_to_reference(oid);
          write_reference(tmp_ref,"Building.ref");
        }
        catch
       (org.omg.PortableServer.POAPackage.ServantAlreadyActive
    ex)
        {
          System.out.println(
              "Caught exception trying to activate an object : "
              + ex.toString()
          );
          System.exit(1);
        }
       catch (org.omg.PortableServer.POAPackage.WrongPolicy ex)
    {
          System.out.println(
              "Caught exception trying to activate an object : "
              + ex.toString()
          );
          System.exit(1);
        }
        catch (
           org.omg.PortableServer.POAPackage.ObjectNotActive ex)
        {
          System.out.println(
              "Caught exception while trying to create reference
    : "
              + ex.toString()
          );
          System.exit(1);
        }
```

```
        // Activate the POA Manager.
        try {
          root_poa_manager.activate();
        }
        catch (
       org.omg.PortableServer.POAManagerPackage.AdapterInactive
   ex)
        {
          System.out.println(
              "Failed trying to activate root poa manager : "
              + ex.toString()
          );
          System.exit(1);
        }

        // Let the ORB process requests
        System.out.println("Waiting for requests..." );
        global_orb.run();
    }
    catch (Exception e) {
        System.out.println(
            "Unexpected CORBA exception: " + e.toString()
        );
    }

    // Ensure that the ORB is properly shutdown and cleaned up
    try  {
        global_orb.shutdown(true);
    }
    catch (Exception e)  {
        // Do nothing.
    }
    return;
  }
}
```

CHAPTER 4

Defining Interfaces

The CORBA Interface Definition Language (IDL) is used to describe interfaces of objects in an enterprise application. An object's interface describes that object to potential clients—its attributes and operations, and their signatures.

An IDL-defined object can be implemented in any language that IDL maps to, such as C++, Java, and COBOL. By encapsulating object interfaces within a common language, IDL facilitates interaction between objects regardless of their actual implementation. Writing object interfaces in IDL is therefore central to achieving the CORBA goal of interoperability between different languages and platforms.

CORBA defines standard mappings from IDL to several programming languages, including C++, Java, and Smalltalk. Each IDL mapping specifies how an IDL interface corresponds to a language-specific implementation. Orbix's IDL compiler uses these mappings to convert IDL definitions to language-specific definitions that conform to the semantics of that language.

This chapter describes IDL semantics and uses. For mapping information, refer to language-specific mappings in the Object Management Group's latest CORBA specification.

In this chapter

This chapter contains the following sections:

Modules and Name Scoping

You create an application's IDL definitions within one or more IDL modules. Each module provides a naming context for the IDL definitions within it.

Modules and interfaces form naming scopes, so identifiers defined inside an interface need to be unique only within that interface. To resolve a name, the IDL compiler conducts its search among the following scopes, in this order:

1. The current interface
2. Base interfaces of the current interface (if any)
3. The scopes that enclose the current interface

In the following example, two interfaces, `Bank` and `Account`, are defined within module `BankDemo`:

```
module BankDemo
{
interface Bank {
        //...
    };

    interface Account {
        //...
    };
};
```

Within the same module, interfaces can reference each other by name alone. If an interface is referenced from outside its module, its name must be fully scoped with the following syntax:

`module-name``::`*`interface-name`*

For example, the fully scoped names of interfaces `Bank` and `Account` are `BankDemo::Bank` and `BankDemo::Account`, respectively.

Nesting restrictions

A module cannot be nested inside a module of the same name. Likewise, you cannot directly nest an interface inside a module of the same name. To avoid name ambiguity, you can provide an intervening name scope as follows:

```
module A
{
    module B
    {
        interface A {
            //...
        };
    };
};
```

Interfaces

Interfaces are the fundamental abstraction mechanism of CORBA. An interface defines a type of object, including the operations that the object supports in a distributed enterprise application.

An IDL interface generally describes an object's behavior through operations and attributes:

- Operations of an interface give clients access to an object's behavior. When a client invokes an operation on an object, it sends a message to that object. The ORB transparently dispatches the call to the object, whether it is in the same address space as the client, in another address space on the same machine, or in an address space on a remote machine.
- An IDL attribute is short-hand for a pair of operations that get and, optionally, set values in an object.

For example, the `Account` interface in module `BankDemo` describes the objects that implement bank accounts:

```
module BankDemo
{
    typedef float CashAmount;  // Type for representing cash
    typedef string AccountId;  // Type for representing account
   ids
    //...
    interface Account {
        readonly attribute AccountId  account_id;
        readonly attribute CashAmount balance;

        void
        withdraw(in CashAmount amount)
        raises (InsufficientFunds);

        void
        deposit(in CashAmount amount);
    };
};
```

This interface declares two readonly attributes, `AccountId` and `balance`, which are defined as typedefs of `string` and `float`, respectively. The interface also defines two operations that a client can invoke on this object, `withdraw()` and `deposit()`.

Because an interface does not expose an object's implementation, all members are public. A client can access variables in an object's implementations only through an interface's operations or attributes.

While every CORBA object has exactly one interface, the same interface can be shared by many CORBA objects in a system. CORBA object references specify CORBA objects—that is, interface instances. Each reference denotes exactly one object, which provides the only means by which that object can be accessed for operation invocations.

Interface Contents

An IDL interface can define the following components:

- Operations
- Attributes
- Exceptions
- Types
- Constants

Of these, operations and attributes must be defined within the scope of an interface; all other components can be defined at a higher scope.

Operations

IDL operations define the signatures of an object's function, which client invocations on that object must use. The signature of an IDL operation is generally composed of three components:

- Return value data type
- Parameters and their direction
- Exception clause

A operation's return value and parameters can use any data types that IDL supports (see "Abstract Interfaces" on page 108).

For example, the `Account` interface defines two operations, `withdraw()` and `deposit()`; it also defines the exception `InsufficientFunds`:

```
module BankDemo
{
    typedef float CashAmount;  // Type for representing cash
    //...
    interface Account {
        exception InsufficientFunds {};

        void
        withdraw(in CashAmount amount)
        raises (InsufficientFunds);

        void
        deposit(in CashAmount amount);
    };
};
```

On each invocation, both operations expect the client to supply an argument for parameter `amount`, and return `void`. Invocations on `withdraw()` can also raise the exception `InsufficientFunds`, if necessary.

Parameter direction

Each parameter specifies the direction in which its arguments are passed between client and object. Parameter passing modes clarify operation definitions and allow the IDL compiler to map operations accurately to a target programming language. At runtime, Orbix uses parameter passing modes to determine in which direction or directions it must marshal a parameter.

A parameter can take one of three passing mode qualifiers:

in: The parameter is initialized only by the client and is passed to the object.

out: The parameter is initialized only by the object and returned to the client.

inout: The parameter is initialized by the client and passed to the server; the server can modify the value before returning it to the client.

In general, you should avoid using `inout` parameters. Because an `inout` parameter automatically overwrites its initial value with a new value, its usage assumes that the caller has no use for the parameter's original value. Thus, the caller must make a copy of the parameter in order to retain that value. By using two parameters, `in` and `out`, the caller can decide for itself when to discard the parameter.

One-way operations

By default, IDL operations calls are *synchronous*—that is, a client invokes an operation on an object and blocks until the invoked operation returns. If an operation definition begins with the keyword `oneway`, a client that calls the operation remains unblocked while the object processes the call.

Three constraints apply to a one-way operation:

- The return value must be set to `void`.
- Directions of all parameters must be set to `in`.
- No `raises` clause is allowed.

For example, interface `Account` might contain a one-way operation that sends a notice to an `Account` object:

```
module BankDemo {
    //...
    interface Account {
        oneway void notice(in string text);
        //...
    };
};
```

Orbix cannot guarantee the success of a one-way operation call. Because one-way operations do not support return data to the client, the client cannot ascertain the outcome of its invocation. Orbix only indicates failure of a one-way operation if the call fails before it exits the client's address space; in this case, Orbix raises a system exception.

Attributes

An interface's attributes correspond to the variables that an object implements. Attributes indicate which variables in an object are accessible to clients.

Unqualified attributes map to a pair of get and set functions in the implementation language, which let client applications read and write attribute values. An attribute that is qualified with the keyword `readonly` maps only to a get function.

For example, the `Account` interface defines two `readonly` attributes, `AccountId` and `balance`. These attributes represent information about the account that only the object implementation can set; clients are limited to read-only access.

Exceptions

IDL operations can raise one or more CORBA-defined system exceptions. You can also define your own exceptions and explicitly specify these in an IDL operation. An IDL exception is a data structure that can contain one or more member fields, formatted as follows:

```
exception exception-name {
   [member;]...
};
```

After you define an exception, you can specify it through a `raises` clause in any operation that is defined within the same scope. A `raises` clause can contain multiple comma-delimited exceptions:

```
return-val operation-name( [params-list] )
    raises( exception-name[, exception-name] );
```

Exceptions that are defined at module scope are accessible to all operations within that module; exceptions that are defined at interface scope are accessible only to operations within that interface.

For example, interface `Account` defines the exception `InsufficientFunds` with a single member of data type `string`. This exception is available to any operation within the interface. The following IDL defines the `withdraw()` operation to raise this exception when the withdrawal fails:

```
module BankDemo
{
    typedef float CashAmount;  // Type for representing cash
    //...
    interface Account {
        exception InsufficientFunds {};

        void
        withdraw(in CashAmount amount)
        raises (InsufficientFunds);
        //...
    };
};
```

For more about exception handling, see Chapter 12 on page 293.

Empty Interfaces

IDL allows you to define empty interfaces. This can be useful when you wish to model an abstract base interface that ties together a number of concrete derived interfaces. For example, the CORBA `PortableServer` module defines the abstract `ServantManager` interface, which serves to join the interfaces for two servant manager types, servant activator and servant locator:

```
module PortableServer
{
    interface ServantManager {};

    interface ServantActivator : ServantManager {
        //...
    };

    interface ServantLocator : ServantManager {
        //...
    };
};
```

Inheritance of IDL Interfaces

An IDL interface can inherit from one or more interfaces. All elements of an inherited, or *base interface*, are available to the *derived interface*. An interface specifies the base interfaces from which it inherits as follows:

```
interface new-interface : base-interface[, base-interface]...
{...};
```

For example, the following interfaces, `CheckingAccount` and `SavingsAccount`, inherit from interface `Account` and implicitly include all of its elements:

```
module BankDemo{
    typedef float CashAmount;  // Type for representing cash
    interface Account {
        //...
    };

    interface CheckingAccount : Account {
        readonly attribute CashAmount overdraftLimit;
        boolean orderCheckBook ();
    };

    interface SavingsAccount : Account {
        float calculateInterest ();
    };
};
```

An object that implements `CheckingAccount` can accept invocations on any of its own attributes and operations and on any of the elements of interface `Account`. However, the actual implementation of elements in a `CheckingAccount` object can differ from the implementation of corresponding elements in an `Account` object. IDL inheritance only ensures type-compatibility of operations and attributes between base and derived interfaces.

Multiple inheritance

The following IDL definition expands module `BankDemo` to include interface `PremiumAccount`, which inherits from two interfaces, `CheckingAccount` and `SavingsAccount`:

```
module BankDemo {
    interface Account {
        //...
    };

    interface CheckingAccount : Account {
        //...
    };

    interface SavingsAccount : Account {
        //...
    };

    interface PremiumAccount :
        CheckingAccount, SavingsAccount {
        //...
    };
};
```

Figure 13 shows the inheritance hierarchy for this interface.

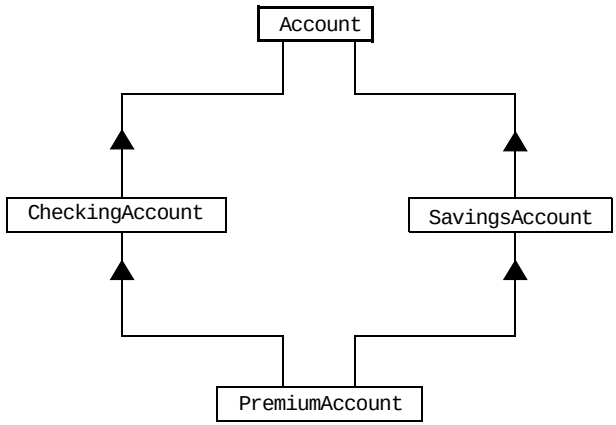


Figure 13: *Multiple inheritance of IDL interfaces*

Multiple inheritance can lead to name ambiguity among elements in the base interfaces. The following constraints apply:

- Names of operations and attributes must be unique across all base interfaces.
- If the base interfaces define constants, types, or exceptions of the same name, references to those elements must be fully scoped.

Inheritance of the object interface

All user-defined interfaces implicitly inherit the predefined interface `Object`. Thus, all `Object` operations can be invoked on any user-defined interface. You can also use `Object` as an attribute or parameter type to indicate that any interface type is valid for the attribute or parameter. For example, the following operation `getAnyObject()` serves as an all-purpose object locator:

```
interface ObjectLocator {
    void getAnyObject (out Object obj);
};
```

Note: It is illegal IDL syntax to inherit interface `Object` explicitly.

Inheritance redefinition

A derived interface can modify the definitions of constants, types, and exceptions that it inherits from a base interface. All other components that are inherited from a base interface cannot be changed. In the following example, interface `CheckingAccount` modifies the definition of exception `InsufficientFunds`, which it inherits from `Account`:

```
module BankDemo
{
    typedef float CashAmount;  // Type for representing cash
    //...
    interface Account {
        exception InsufficientFunds {};
        //...
    };
    interface CheckingAccount : Account {
        exception InsufficientFunds {
            CashAmount overdraftLimit;
        };
    };
    //...
};
```

Note: While a derived interface definition cannot override base operations or attributes, operation overloading is permitted in interface implementations for those languages such as C++ that support it.

Forward Declaration of IDL Interfaces

An IDL interface must be declared before another interface can reference it. If two interfaces reference each other, the module must contain a forward declaration for one of them; otherwise, the IDL compiler reports an error. A forward declaration only declares the interface's name; the interface's actual definition is deferred until later in the module.

For example, IDL interface `Bank` defines two operations that return references to `Account` objects—`create_account()` and `find_account()`. Because interface `Bank` precedes the definition of interface `Account`, `Account` is forward-declared as follows:

```
module BankDemo
{
    typedef float CashAmount;  // Type for representing cash
    typedef string AccountId;  // Type for representing account
   ids

    // Forward declaration of Account
    interface Account;

    // Bank interface...used to create Accounts
    interface Bank {
        exception AccountAlreadyExists { AccountId account_id; };
        exception AccountNotFound      { AccountId account_id; };

        Account
        find_account(in AccountId account_id)
        raises(AccountNotFound);

        Account
        create_account(
            in AccountId account_id,
            in CashAmount initial_balance
        ) raises (AccountAlreadyExists);
    };

    // Account interface...used to deposit, withdraw, and query
    // available funds.
    interface Account {
        //...
    };
};
```

Local Interfaces

An interface declaration that contains the keyword `local` defines a *local interface*. An interface declaration that omits this keyword can be referred to as an *unconstrained interface*, to distinguish it from local interfaces. An object that implements a local interface is a *local object*.

Local interfaces differ from unconstrained interfaces in the following ways:

- A local interface can inherit from any interface, whether local or unconstrained. However, an unconstrained interface cannot inherit from a local interface.
- Any non-interface type that uses a local interface is regarded as a *local type*. For example, a struct that contains a local interface member is regarded as a local struct, and is subject to the same localization constraints as a local interface.
- Local types can be declared as parameters, attributes, return types, or exceptions only in a local interface, or as state members of a valuetype.
- Local types cannot be marshaled, and references to local objects cannot be converted to strings through `ORB::object_to_string()`. Attempts to do so throw `CORBA::MARSHAL`.
- Any operation that expects a reference to a remote object cannot be invoked on a local object. For example, you cannot invoke any DII operations or asynchronous methods on a local object; similarly, you cannot invoke pseudo-object operations such as `is_a()` or `validate_connection()`. Attempts to do so throw `CORBA::NO_IMPLEMENT`.
- The ORB does not mediate any invocation on a local object. Thus, local interface implementations are responsible for providing the parameter copy semantics that a client expects.
- Instances of local objects that the OMG defines as supplied by ORB products are exposed either directly or indirectly through `ORB::resolve_initial_references()`.

Local interfaces are implemented by `CORBA::LocalObject` to provide implementations of Object pseudo operations, and other ORB-specific support mechanisms that apply. Because object implementations are language-specific, the `LocalObject` type is only defined by each language mapping.

The `LocalObject` type implements the following Object pseudo-operations to throw an exception of `NO_IMPLEMENT`:

```
is_a()
get_interface()
get_domain_managers()
get_policy()
get_client_policy()
set_policy_overrides()
get_policy_overrides()
validate_connection()
```

`CORBA::LocalObject` also implements the pseudo-operations shown in Table 1:

Table 1: *CORBA::LocalObject pseudo-operation returns*

Operation	Always returns:
`non_existent()`	False
`hash()`	A hash value that is consistent with the object's lifetime
`is_equivalent()`	True if the references refer to the same `LocalObject` implementation.

Valuetypes

Valuetypes enable programs to pass objects by value across a distributed system. This type is especially useful for encapsulating lightweight data such as linked lists, graphs, and dates.

Valuetypes can be seen as a cross between data types such as `long` and `string` that can be passed by value over the wire as arguments to remote invocations, and objects, which can only be passed by reference. When a program supplies an object reference, the object remains in its original location; subsequent invocations on that object from other address spaces move across the network, rather than the object moving to the site of each request.

Like an interface, a valuetype supports both operations and inheritance from other valuetypes; it also can have data members. When a valuetype is passed as an argument to a remote operation, the receiving address space creates a copy it of it. The copied valuetype exists independently of the original; operations that are invoked on one have no effect on the other.

Because a valuetype is always passed by value, its operations can only be invoked locally. Unlike invocations on objects, valuetype invocations are never passed over the wire to a remote valuetype.

Valuetype implementations necessarily vary, depending on the languages used on sending and receiving ends of the transmission, and their respective abilities to marshal and demarshal the valuetype's operations. A receiving process that is written in C++ must provide a class that implements valuetype operations and a factory to create instances of that class. These classes must be either compiled into the application, or made available through a shared library. Conversely, Java applications can marshal enough information on the sender, so the receiver can download the bytecodes for the valuetype operation implementations.

Abstract Interfaces

An application can use abstract interfaces to determine at runtime whether an object is passed by reference or by value. For example, the following IDL definitions specify that operation `Example::display()` accepts any derivation of abstract interface `Describable`:

```
abstract interface Describable {
    string get_description();
};

interface Example {
    void display(in Describable someObject);
};
```

Given these definitions, you can define two derivations of abstract interface `Describable`, valuetype `Currency` and interface `Account`:

```
interface Account : Describable {
    // body of Account definition not shown
};

valuetype Currency supports Describable {
    // body of Currency definition not shown
};
```

Because the parameter for `display()` is defined as a `Describable` type, invocations on this operation can supply either `Account` objects or `Currency` valuetypes.

All abstract interfaces implicitly inherit from native type `CORBA::AbstractBase`, and map to Java interfaces. Abstract interfaces have several characteristics that differentiate them from interfaces:

- The GIOP encoding of an abstract interface contains a boolean discriminator to indicate whether the adjoining data is an IOR (`TRUE`) or a value (`FALSE`). The demarshalling code can thus determine whether the argument passed to it is an object reference or a value.

- Unlike interfaces, abstract interfaces do not inherit from `CORBA::Object`, in order to allow support for valuetypes. If the runtime argument supplied to an abstract interface type can be narrowed to an object reference type, then `CORBA::Object` operations can be invoked on it.
- Because abstract interfaces can be derived by object references or by value types, copy semantics cannot be guaranteed for value types that are supplied as arguments to its operations.
- Abstract interfaces can only inherit from other abstract interfaces.

IDL Data Types

In addition to IDL module, interface, valuetype, and exception types, IDL data types can be grouped into the following categories:

- Built-in types such as `short`, `long`, and `float`
- Extended built-in types such as `long long` and `wstring`
- Complex data types such as `enum` and `struct`, and `string`
- Pseudo object types

Built-in Types

Table 2 lists built-in IDL types.

Table 2: *Built-in IDL types*

Data type	Size	Range of values
short	16 bits	$-2^{15}...2^{15}-1$
unsigned short	16 bits	$0...2^{16}-1$
long	32 bits	$-2^{31}...2^{31}-1$
unsigned long	32 bits	$0...2^{32}-1$
float	32 bits	IEEE single-precision floating point numbers
double	64 bits	IEEE double-precision floating point numbers
char	8 bits	ISO Latin-1
string	variable length	ISO Latin-1, except NUL
string<bound>	variable length	ISO Latin-1, except NUL
boolean	unspecified	TRUE or FALSE
octet	8 bits	0x0 to 0xff
any	variable length	Universal container type

Integer types

IDL supports short and long integer types, both signed and unsigned. IDL guarantees the range of these types. For example, an unsigned short can hold values between 0-65535. Thus, an unsigned short value always maps to a native type that has at least 16 bits. If the platform does not provide a native 16-bit type, the next larger integer type is used.

Floating point types

Types float and double follow IEEE specifications for single- and double-precision floating point values, and on most platforms map to native IEEE floating point types.

char

Type `char` can hold any value from the ISO Latin-1 character set. Code positions 0-127 are identical to ASCII. Code positions 128-255 are reserved for special characters in various European languages, such as accented vowels.

String types

Type `string` can hold any character from the ISO Latin-1 character set except `NUL`. IDL prohibits embedded `NUL` characters in strings. Unbounded string lengths are generally constrained only by memory limitations. A bounded string, such as `string<10>`, can hold only the number of characters specified by the bounds, excluding the terminating `NUL` character. Thus, a `string<6>` can contain the six-character string `cheese`.

The declaration statement can optionally specify the string's maximum length, thereby determining whether the string is bounded or unbounded:

```
string[<length>] name
```

For example, the following code declares data type `ShortString`, which is a bounded string whose maximum length is 10 characters:

```
typedef string<10> ShortString;
attribute ShortString shortName; // max length is 10 chars
```

octet

`Octet` types are guaranteed not to undergo any conversions in transit. This lets you safely transmit binary data between different address spaces. Avoid using type `char` for binary data, inasmuch as characters might be subject to translation during transmission. For example, if client that uses ASCII sends a string to a server that uses EBCDIC, the sender and receiver are liable to have different binary values for the string's characters.

any

Type `any` allows specification of values that express any IDL type, which is determined at runtime. An `any` logically contains a `TypeCode` and a value that is described by the `TypeCode`. For more information about the `any` data type, see Chapter 14 on page 325.

Extended Built-in Types

Table 3 lists extended built-in IDL types.

Table 3: *Extended built-in IDL types*

Data type	Size	Range of values
`long long`	64 bits	$-2^{63}...2^{63}-1$
`unsigned long long`	64 bits	$0...-2^{64}-1$
`long double`	79 bits	IEEE double-extended floating point number, with an exponent of at least 15 bits in length and signed fraction of at least 64 bits. `long double` type is currently not supported on Windows NT.
`wchar`	Unspecified	Arbitrary codesets
`wstring`	Variable length	Arbitrary codesets
`fixed`	Unspecified	31 significant digits

long long

The 64-bit integer types `long long` and `unsigned long long` support numbers that are too large for 32-bit integers. Platform support varies. If you compile IDL that contains one of these types on a platform that does not support it, the compiler issues an error.

long double

Like 64-bit integer types, platform support varies for `long double`, so usage can yield IDL compiler errors.

wchar

Type `wchar` encodes wide characters from any character set. The size of a `wchar` is platform-dependent.

wstring

Type `wstring` is the wide-character equivalent of type `string` (see page 112). Like `string`-types, `wstring` types can be unbounded or bounded. Wide strings can contain any character except `NUL`.

fixed

Type `fixed` provides fixed-point arithmetic values with up to 31 significant digits. You specify a `fixed` type with the following format:

`typedef fixed< digit-size, scale > name`

`digit-size` specifies the number's length in digits. The maximum value for *`digit-size`* is 31 and must be greater than *`scale`*. A fixed type can hold any value up to the maximum value of a `double`.

Scaling options

If *`scale`* is a positive integer, it specifies where to place the decimal point relative to the rightmost digit. For example the following code declares fixed data type `CashAmount` to have a digit size of `8` and a scale of `2`:

```
typedef fixed<10,2> CashAmount;
```

Given this typedef, any variable of type `CashAmount` can contain values of up to (+/-)99999999.99.

If *`scale`* is negative, the decimal point moves to the right *`scale`* digits, thereby adding trailing zeros to the fixed data type's value. For example, the following code declares fixed data type `bigNum` to have a digit size of `3` and a scale of `-4`:

```
typedef fixed <3,-4> bigNum;
bigNum myBigNum;
```

If `myBigNum` has a value of `123`, its numeric value resolves to `1230000`. Definitions of this sort let you store numbers with trailing zeros efficiently.

Constant fixed types

Constant fixed types can also be declared in IDL, where *`digit-size`* and *`scale`* are automatically calculated from the constant value. For example:

```
module Circle {
    const fixed pi = 3.142857;
};
```

This yields a fixed type with a digit size of `7`, and a scale of `6`.

Unlike IEEEE floating-point values, type `fixed` is not subject to representational errors. IEEE floating point values are liable to represent decimal fractions inaccurately unless the value is a fractional power of 2.

For example, the decimal value 0.1 cannot be represented exactly in IEEE format. Over a series of computations with floating-point values, the cumulative effect of this imprecision can eventually yield inaccurate results.

Type `fixed` is especially useful in calculations that cannot tolerate any imprecision, such as computations of monetary values.

Complex Data Types

IDL provides the following complex data types:

- enum
- struct
- union
- multi-dimensional fixed-size arrays
- sequence

enum

An enum (enumerated) type lets you assign identifiers to the members of a set of values. For example, you can modify the `BankDemo` IDL with enum type `balanceCurrency`:

```
module BankDemo {
    enum Currency {pound, dollar, yen, franc};

    interface Account {
        readonly attribute CashAmount balance;
        readonly attribute Currency balanceCurrency;
        //...
    };
};
```

In this example, attribute `balanceCurrency` in interface `Account` can take any one of the values `pound`, `dollar`, `yen`, or `franc`.

The actual ordinal values of a `enum` type vary according to the actual language implementation. The CORBA specification only guarantees that the ordinal values of enumerated types monotonically increase from left to right. Thus, in the previous example, `dollar` is greater than `pound`, `yen` is greater than `dollar`, and so on. All enumerators are mapped to a 32-bit type.

struct

A `struct` data type lets you package a set of named members of various types. In the following example, struct `CustomerDetails` has several members. Operation `getCustomerDetails()` returns a struct of type `CustomerDetails` that contains customer data:

```
module BankDemo{
        struct CustomerDetails {
            string custID;
            string lname;
            string fname;
            short age;
            //...
        };

        interface Bank {
            CustomerDetails getCustomerDetails
                (in string custID);
            //...
        };
};
```

A `struct` must include at least one member. Because a `struct` provides a naming scope, member names must be unique only within the enclosing structure.

union

A union data type lets you define a structure that can contain only one of several alternative members at any given time. A union saves space in memory, as the amount of storage required for a union is the amount necessary to store its largest member.

You declare a union type with the following syntax:

```
union name switch (discriminator) {
    case label1 : element-spec;
    case label2 : element-spec;
    [...]
    case labeln : element-spec;
    [default : element-spec;]
};
```

All IDL unions are *discriminated*. A discriminated union associates a constant expression (*label1..labeln*) with each member. The discriminator's value determines which of the members is active and stores the union's value.

For example, the following code defines the IDL union `Date`, which is discriminated by an `enum` value:

```
enum dateStorage
{ numeric, strMMDDYY, strDDMMYY };

struct DateStructure {
    short Day;
    short Month;
    short Year;
};

union Date switch (dateStorage) {
    case numeric: long digitalFormat;
    case strMMDDYY:
    case strDDMMYY: string stringFormat;
    default: DateStructure structFormat;
};
```

Given this definition, if `Date`'s discriminator value is `numeric`, then `digitalFormat` member is active; if the discriminator's value is `strMMDDYY or strDDMMYY`, then member `stringFormat` is active; otherwise, the default member `structFormat` is active.

The following rules apply to `union` types:

- A union's discriminator can be `integer`, `char`, `boolean` or `enum`, or an alias of one of these types; all `case` label expressions must be compatible with this type.
- Because a `union` provides a naming scope, member names must be unique only within the enclosing union.
- Each `union` contains a pair of values: the discriminator value and the active member.
- IDL unions allow multiple case labels for a single member. In the previous example, member `stringFormat` is active when the discriminator is either `strMMDDYY or strDDMMYY`.

- IDL unions can optionally contain a `default` case label. The corresponding member is active if the discriminator value does not correspond to any other label.

---

arrays

IDL supports multi-dimensional fixed-size arrays of any IDL data type, with the following syntax:

```
[typedef] element-type array-name [dimension-spec]...
```

dimension-spec must be a non-zero positive constant integer expression. IDL does not allow open arrays. However, you can achieve equivalent functionality with sequence types (see page 119).

For example, the following code fragment defines a two-dimensional array of bank accounts within a portfolio:

```
typedef Account portfolio[MAX_ACCT_TYPES][MAX_ACCTS]
```

An array must be named by a `typedef` declaration (see "Defining Data Types" on page 122) in order to be used as a parameter, an attribute, or a return value. You can omit a `typedef` declaration only for an array that is declared within a structure definition.

Because of differences between implementation languages, IDL does not specify the origin at which arrays are indexed. For example C and C++ array indexes always start at 0, while Pascal uses an origin of 1. Consequently, clients and servers cannot portably exchange array indexes unless they both agree on the origin of array indexes and make adjustments as appropriate for their respective implementation languages. Usually, it is easier to exchange the array element itself instead of its index.

---

sequence

IDL supports sequences of any IDL data type with the following syntax:

```
[typedef] sequence < element-type[, max-elements] >
   sequence-name
```

An IDL sequence is similar to a one-dimensional array of elements; however, its length varies according to its actual number of elements, so it uses memory more efficiently.

A sequence must be named by a `typedef` declaration (see "Defining Data Types" on page 122) in order to be used as a parameter, an attribute, or a return value. You can omit a `typedef` declaration only for a sequence that is declared within a structure definition.

A sequence's element type can be of any type, including another sequence type. This feature is often used to model trees.

The maximum length of a sequence can be fixed (bounded) or unfixed (unbounded):

- Unbounded sequences can hold any number of elements, up to the memory limits of your platform.
- Bounded sequences can hold any number of elements, up to the limit specified by the bound.

The following code shows how to declare bounded and unbounded sequences as members of an IDL struct:

```
struct LimitedAccounts {
    string bankSortCode<10>;
    sequence<Account, 50> accounts; // max sequence length is 50
};

struct UnlimitedAccounts {
    string bankSortCode<10>;
    sequence<Account> accounts; // no max sequence length
};
```

Pseudo Object Types

CORBA defines a set of pseudo object types that ORB implementations use when mapping IDL to a programming language. These object types have interfaces defined in IDL but do not have to follow the normal IDL mapping for interfaces and are not generally available in your IDL specifications.

You can use only the following pseudo object types as attribute or operation parameter types in an IDL specification:

```
CORBA::NamedValue
CORBA::TypeCode
```

To use these types in an IDL specification, include the file `orb.idl` in the IDL file as follows:

```
#include <orb.idl>
//...
```

This statement tells the IDL compiler to allow types `NamedValue` and `TypeCode`.

Defining Data Types

With `typedef`, you can define more meaningful or simpler names for existing data types, whether IDL-defined or user-defined. The following IDL defines `typedef` identifier `StandardAccount`, so it can act as an alias for type `Account` in later IDL definitions:

```
module BankDemo {
    interface Account {
        //...
    };

    typedef Account StandardAccount;
};
```

Constants

IDL lets you define constants of all built-in types except type any. To define a constant's value, you can either use another constant (or constant expression) or a literal. You can use a constant wherever a literal is permitted.

The following constant types are supported:

- Integer
- Floating-point
- Character and string
- Wide character and string
- Boolean
- Octet
- Fixed-point
- Enumeration

Integer

IDL accepts integer literals in decimal, octal, or hexadecimal:

```
const short     I1 = -99;
const long      I2 = 0123;  // Octal 123, decimal 83
const long long I3 = 0x123; // Hexadecimal 123, decimal 291
const long long I4 = +0xaB; // Hexadecimal ab, decimal 171
```

Both unary plus and unary minus are legal.

Floating-point

Floating-point literals use the same syntax as C++:

```
const float       f1 = 3.1e-9; // Integer part, fraction part,
                               // exponent
const double      f2 = -3.14;  // Integer part and fraction part
const long double f3 = .1      // Fraction part only
const double      f4 = 1.      // Integer part only
const double      f5 = .1E12   // Fraction part and exponent
const double      f6 = 2E12    // Integer part and exponent
```

Character and string

Character constants use the same escape sequences as C++:

```
const char C1 = 'c';        // the character c
const char C2 = '\007';     // ASCII BEL, octal escape
const char C3 = '\x41';     // ASCII A, hex escape
const char C4 = '\n';       // newline
const char C5 = '\t';       // tab
const char C6 = '\v';       // vertical tab
const char C7 = '\b';       // backspace
const char C8 = '\r';       // carriage return
const char C9 = '\f';       // form feed
const char C10 = '\a';      // alert
const char C11 = '\\';      // backslash
const char C12 = '\?';      // question mark
const char C13 = '\'';      // single quote
// String constants support the same escape sequences as C++
const string S1 = "Quote: \"";      // string with double quote
const string S2 = "hello world";    // simple string
const string S3 = "hello" " world"; // concatenate
const string S4 = "\xA" "B";        // two characters
                                    // ('\xA' and 'B'),
                                    // not the single character '\xAB
```

Wide character and string

Wide character and string constants use C++ syntax. Use Universal character codes to represent arbitrary characters. For example:

```
const wchar     C = L'X';
const wstring   GREETING = L"Hello";
const wchar     OMEGA = L'\u03a9';
const wstring   OMEGA_STR = L"Omega: \u3A9";
```

Note: IDL files themselves always use the ISO Latin-1 code set, they cannot use Unicode or other extended character sets.

Boolean

Boolean constants use the keywords FALSE and TRUE. Their use is unnecessary, inasmuch as they create needless aliases:

```
// There is no need to define boolean constants:
const CONTRADICTION = FALSE;    // Pointless and confusing
const TAUTOLOGY = TRUE;         // Pointless and confusing
```

Octet

Octet constants are positive integers in the range 0-255.

```
const octet O1 = 23;
const octet O2 = 0xf0;
```

Note: Octet constants were added with CORBA 2.3, so ORBs that are not compliant with this specification might not support them.

Fixed-point

For fixed-point constants, you do not explicitly specify the digits and scale. Instead, they are inferred from the initializer. The initializer must end in `d` or `D`. For example:

```
// Fixed point constants take digits and scale from the
// initialiser:
const fixed val1 = 3D;          // fixed<1,0>
const fixed val2 = 03.14d;      // fixed<3,2>
const fixed val3 = -03000.00D;  // fixed<4,0>
const fixed val4 = 0.03D;       // fixed<3,2>
```

The type of a fixed-point constant is determined after removing leading and trailing zeros. The remaining digits are counted to determine the digits and scale. The decimal point is optional.

Note: Currently, there is no way to control the scale of a constant if it ends in trailing zeros.

Enumeration

Enumeration constants must be initialized with the scoped or unscoped name of an enumerator that is a member of the type of the enumeration. For example:

```
enum Size { small, medium, large };

const Size DFL_SIZE = medium;
const Size MAX_SIZE = ::large;
```

Note: Enumeration constants were added with CORBA 2.3, so ORBs that are not compliant with this specification might not support them.

Constant Expressions

IDL provides a number of arithmetic and bitwise operators.

Operator precedence

The precedence for operators follows the rules for C++. You can override the default precedence by adding parentheses.

Arithmetic operators

The arithmetic operators have the usual meaning and apply to integral, floating-point, and fixed-point types (except for %, which requires integral operands). However, these operators do not support mixed-mode arithmetic; you cannot, for example, add an integral value to a floating-point value. The following code contains several examples:

```
// You can use arithmetic expressions to define constants.
const long MIN = -10;
const long MAX = 30;
const long DFLT = (MIN + MAX) / 2;

// Can't use 2 here
const double TWICE_PI = 3.1415926 * 2.0;

// 5% discount
const fixed DISCOUNT = 0.05D;
const fixed PRICE = 99.99D;

// Can't use 1 here
const fixed NET_PRICE = PRICE * (1.0D - DISCOUNT);
```

Expressions are evaluated using the type promotion rules of C++. The result is coerced back into the target type. The behavior for overflow is undefined, so do not rely on it. Fixed-point expressions are evaluated internally with 62 bits of precision, and results are truncated to 31 digits.

Bitwise operators

The bitwise operators only apply to integral types. The right-hand operand must be in the range 0–63. Note that the right-shift operator >> is guaranteed to inject zeros on the left, whether the left-hand operand is signed or unsigned:

```
// You can use bitwise operators to define constants.
const long ALL_ONES = -1;              // 0xffffffff
const long LHW_MASK = ALL_ONES << 16;  // 0xffff0000
const long RHW_MASK = ALL_ONES >> 16;  // 0x0000ffff
```

IDL guarantees two's complement binary representation of values.

CHAPTER 5

Developing Applications with Genies

The code generation toolkit is packaged with a genie that can help your development effort get off to a fast start.

The Java genie `java_poa_genie.tcl` creates a complete, working client and server directly from your IDL interfaces. You can then add application logic to the generated code. This can improve productivity in two ways:

- The outlines of your application—class declarations and operation signatures—are generated for you.
- A working system is available immediately, which you can incrementally modify and test. With the generated build files, you can build and test modifications right away, thereby eliminating much of the overhead that is usually associated with getting a new project underway.

In a genie-generated application, the client invokes every operation and each attribute's get and set methods, and directs all display to standard output. The server also writes all called operations to standard output.

This client/server application achieves these goals:

- Demonstrates or tests an Orbix client/server application for a particular interface or interfaces.

- Provides a starting point for your application.
- Shows the right way to initialize and pass parameters for various IDL data types.

In this chapter

This chapter covers the following topics:

Genie Syntax

`java_poa_genie.tcl` uses the following syntax:

`idlgen java_poa_genie.tcl` *`component-spec [options] idl-file`*

You must specify an IDL file. You must also specify the application components to generate, either all components at once, or individual components, with one of the arguments in Table 4:

Table 4: *Component specifier arguments to java_poa_genie.tcl*

Component specifier	Output
`-all`	All components: server, servant, client, and antfile (see page 133).
`-servant`	Servant classes to implement the selected interfaces (see page 137).
`-server`	Server main program (see page 140)
`-client`	Client main program (see page 143).
`-antfile`	Files used by the `itant` utility to compile server and client applications (see page 144).

Each component specifier can take its own arguments. For more information on these, refer to the discussion on each component later in this chapter.

You can also supply one or more of the optional switches shown in Table 5:

Table 5: *Optional switches to java_poa_genie.tcl*

Option	Description
`-complete/-incomplete`	Controls the completeness of the code that is generated for the specified components (see page 145).
`-dir`	Specifies where to generate file output (see page 151).

Table 5: *Optional switches to java_poa_genie.tcl*

Option	Description
`-include`	Specifies to generate code for included files (see page 136).
`-interface-spec`	Specifies to generate code only for the specified interfaces (see page 135).
`-jPpackage-name`	Specifies the package name to contain the file output (see page 151).
`-v/-s`	Controls the level of verbosity (see page 151).

Specifying Application Components

The `-all` argument generates the files that implement all application components: server, servant, client, and build files. For example, the following command generates all the files required for an application that is based on `bankdemo.idl`:

```
> idlgen java_poa_genie.tcl -all bankdemo.idl

java_poa_genie.tcl: creating idlgen/RandomFuncs.java
java_poa_genie.tcl: creating
   idlgen/NoPackage/RandomBankDemo.java
java_poa_genie.tcl: creating
   idlgen/NoPackage/BankDemo/RandomBank.java
java_poa_genie.tcl: creating
   idlgen/NoPackage/BankDemo/RandomAccount.java
java_poa_genie.tcl: creating idlgen/RandomNoPackage.java
java_poa_genie.tcl: creating NoPackage/BankDemo/BankCaller.java
java_poa_genie.tcl: creating
   NoPackage/BankDemo/AccountCaller.java
java_poa_genie.tcl: creating NoPackage/client.java
java_poa_genie.tcl: creating NoPackage/BankDemo/BankImpl.java
java_poa_genie.tcl: creating NoPackage/BankDemo/AccountImpl.java
java_poa_genie.tcl: creating NoPackage/server.java
java_poa_genie.tcl: creating build.xml
```

Alternatively, you can use `java_poa_genie.tcl` to generate one or more application components. For example, the following command specifies to generate only those files that are required to implement a server:

```
> idlgen java_poa_genie.tcl -server bankdemo.idl

java_poa_genie.tcl: creating idlgen/PrintFuncs.java
java_poa_genie.tcl: creating idlgen/NoPackage/PrintBankDemo.java
java_poa_genie.tcl: creating
   idlgen/NoPackage/BankDemo/PrintBank.java
java_poa_genie.tcl: creating
   idlgen/NoPackage/BankDemo/PrintAccount.java
java_poa_genie.tcl: creating idlgen/RandomFuncs.java
java_poa_genie.tcl: creating
   idlgen/NoPackage/RandomBankDemo.java
java_poa_genie.tcl: creating
   idlgen/NoPackage/BankDemo/RandomBank.java
java_poa_genie.tcl: creating
   idlgen/NoPackage/BankDemo/RandomAccount.java
java_poa_genie.tcl: creating idlgen/RandomNoPackage.java
java_poa_genie.tcl: creating NoPackage/server.java
```

By generating output for application components selectively, you can control genie processing for each one. For example, the following commands specify different `-dir` options, so that server and servant files are output to one directory, and client files are output to another:

```
> idlgen java_poa_genie.tcl -servant - server bankdemo.idl
    -dir c:\app\server
> idlgen java_poa_genie.tcl -client bankdemo.idl
    -dir c:\app\client
```

Selecting Interfaces

By default, `java_poa_genie.tcl` generates code for all interfaces in the specified IDL file. You can specify to generate code for specific interfaces within the file by supplying their fully scoped names. For example, the following command specifies to generate code for the Bank interface in `bankdemo.idl`:

```
> idlgen java_poa_genie.tcl -all BankDemo::Bank bankdemo.idl
```

You can also use wildcard patterns to specify the interfaces to process. For example, the following command generates code for all interfaces in module `BankDemo`:

```
> idlgen java_poa_genie.tcl BankDemo::* bankdemo.idl
```

The following command generates code for all interfaces in `foo.idl` with names that begin with `Foo` or end with `Bar`.

```
> idlgen java_poa_genie.tcl foo.idl "Foo*" "*Bar"
```

Note: For interfaces defined inside modules, the wildcard is matched against the fully scoped interface name, so `Foo*` matches `FooModule::Y` but not `BarModule::Foo`.

Pattern matching is performed according to the rules of the TCL `string match` command, which is similar to Unix or Windows filename matching. Table 6 contains some common wildcard patterns:

Table 6: *Wildcard pattern matching to interface names*

Wildcard pattern	Matches...
*	Any string
?	Any single character
[xyz]	x, y, or z.

Including Files

By default, `java_poa_genie.tcl` generates code only for the specified IDL files. You can specify also to generate code for all `#include` files by supplying the `-include` option. For example, the following command specifies to generate code from `bankdemo.idl` and any IDL files that are included in it:

```
> idlgen java_poa_genie.tcl -all -include bankdemo.idl
```

The default for this option is set in the configuration file through `default.java_genie.want_include`.

Implementing Servants

The `-servant` option generates POA servant classes that implement IDL interfaces. For example, this command generates a class header and implementation code for each interface that appears in IDL file `bankdemo.idl`:

```
idlgen java_poa_genie.tcl -servant bankdemo.idl
```

The genie constructs the implementation class name by adding a suffix—by default, `Impl`—to the interface name. The default suffix is set in the configuration file through `default.java.impl_class_suffix`.

For example, `BankDemo::Account` is implemented by class `AccountImpl`. The generated implementation class contains these components:

- A static `_create()` member method to create a servant.
- A member method to implement each IDL operation for the interface.

The `-servant` option can take one or more arguments, shown in Table 7, that let you control how servant classes are generated:

Table 7: *Arguments that control servant generation*

Argument	Purpose
`-tie` `-notie`	Choose the inheritance or tie (delegation) method for implementing servants.
`-inherit` `-noinherit`	Choose whether implementation classes follow the same inheritance hierarchy as the IDL interfaces they implement.
`-default_poa` *arg*	Determines the behavior of implicit activation, which uses the default POA associated with a given servant. `default_poa` can take one of these arguments: • `per_servant`: Set the correct default POA for each servant. • `exception`: Throw an exception on all attempts at implicit activation. For more information, see page 214.

The actual content and behavior of member methods is determined by the `-complete` or `-incomplete` flag. For more information, see "Controlling Code Completeness" on page 145.

-tie/-notie

A POA servant is either an instance of a class that inherits from a POA skeleton, or an instance of a tie template class that delegates to a separate implementation class. You can choose the desired approach by supplying `-tie` or `-notie` options. The default for this option is set in the configuration file through `default.java_genie.want_tie`.

With `-notie`, the genie generates servants that inherit directly from POA skeletons. For example:

`public class AccountImpl extends AccountPOA`

The `_create()` method constructs a servant as follows:

```
public static AccountImpl _create(
    org.omg.PortableServer.POA the_poa)
    throws org.omg.CORBA.SystemException
{
      return new AccountImpl(the_poa);
}
```

With `-tie`, the genie generates implementation classes that do not inherit from POA skeletons. The following example uses a `_create` method to create an implementation object (1), and a tie (2) that delegates to it:

Example 11: *Java Creating a TIE Object*

```
public static NoPackage.BankDemo.AccountPOATie _create(
    org.omg.PortableServer.POA the_poa)
    throws org.omg.CORBA.SystemException
{
      AccountImpl tied_object = new AccountImpl();
      NoPackage.BankDemo.AccountPOATie the_tie =
          new NoPackage.BankDemo.AccountPOATie(
          tied_object, the_poa);

      return the_tie;
}
```

Note: `_create()` is a useful genie convention that provides a consistent way to create servants whether you use the tie approach or not. This helps minimize the impact on your code if you change approaches during development. You can also create servants and tie objects by calling the constructors directly in your own code.

-inherit/-noinherit

IDL servant implementation classes typically have the same inheritance hierarchy as the interfaces that they implement, but this is not required.

- `-inherit` generates implementation classes with the same inheritance as the corresponding interfaces.
- `-noinherit` generates implementation classes that do not inherit from each other. Instead, each implementation class independently implements all operations for its IDL interface, including operations that are inherited from other IDL interfaces.

The default for this option is set in the configuration file through `default.java_genie.want_inherit`.

-default_poa

In the standard CORBA Java mapping, each servant class provides a `_this()` method, which generates an object reference and implicitly activates that object with the servant. Implicit activation calls `_default_POA()` on the same servant to determine the POA in which this object is activated. Unless you specify otherwise, `_default_POA()` returns the root POA, which is typically not the POA where you want to activate objects.

The code that `java_poa_genie.tcl` generates always overrides `_default_POA()` in a way that prevents implicit activation. Applications generated by this genie can only activate objects explicitly. Two options are available that determine how to override `_default_POA()`:

- `per_servant`: (default) Servant constructors and generated `_create()` methods takes a POA parameter. For each servant, `_default_POA()` returns the POA specified when the servant was created.
- `exception`: `_default_POA()` throws a `CORBA::INTERNAL` system exception. This option is useful in a group development environment, in that it allows tests to easily catch any attempts at implicit activation.

For more information about explicit and implicit activation, see page 213.

Implementing the Server Mainline

The `-server` option generates a simple server mainline that activates and exports some objects. For example, the following command generates a file called `serverjava` that contains a `main` program:

```
> idlgen java_poa_genie.tcl -server bankdemo.idl
```

The server program performs the following steps:

1. Initializes the ORB and POA.
2. For each interface:
   - Activates a CORBA object of that interface.
   - Exports a reference either to the naming service or to a file, depending on whether you set the option `-ns` or `-nons`.
3. Catches any exceptions and print a message.

The `-server` option can take one or more arguments, shown in Table 8, that let you modify server behavior:

Table 8: *Options affecting the server*

Command line option	Purpose
-threads -nothreads	Choose a single or multi-threaded server.
-strategy simple	Create servants during start-up.
-strategy activator	Create servants on demand with a servant activator.
-strategy locator	Create servants per call with a servant locator.
-strategy default_servant	For each interface, generate a POA that uses a default servant.
-ns -nons	Determines how to export object references: • `-ns`: use the naming service to publish object references. • `-nons`: write object references to a file.

-threads/-nothreads

You can specify the threads policy for all POAs in the server with one of these options:

-nothreads sets the `SINGLE_THREAD_MODEL` policy on all POAs in the server, which ensures that all calls to application code are made in the main thread. This policy allows a server to run thread-unsafe code, but might reduce performance because the ORB can dispatch only one operation at a time.

-threads sets the `ORB_CTRL_MODEL` policy on all POAs in the server, allowing the ORB to dispatch incoming calls in multiple threads concurrently.

Note: If you enable multi-threading, you must ensure that your application code is thread-safe and application data structures are adequately protected by thread-synchronization calls.

The default for this option is set in the configuration file through `default.java_genie.want_threads`.

-strategy Options

The POA is a flexible tool that lets servers manage objects with different strategies. Some servers can use a combination of strategies for different objects. You can use the genie to generate examples of each strategy, then cut-and-paste the appropriate generated code into your own server.

You set a server's object management strategy through one of the following arguments to the `-strategy` option:

-strategy simple: The server creates a POA with a policy of `USE_ACTIVE_OBJECT_MAP_ONLY` (see page 204). For each interface in the IDL file, the server `main()` creates a servant, activates it with the POA as a CORBA object, and exports an object reference.

This strategy is appropriate for servers that implement a small, fixed set of objects.

-strategy activator: The server creates a POA and a servant activator (see "Servant Activators" on page 276). For each interface, the server exports an object reference. The object remains inactive until a client first calls on its reference; then, the servant activator is invoked and creates the appropriate servant, which remains in memory to handle future calls on that reference.

This strategy lets the server start receiving requests immediately and defer creation of servants until they are needed. It is useful for servers that normally activate just a few objects out of a large collection on each run, or for servants that take a long time to initialize.

-strategy locator: The server creates a POA and a servant locator (see "Servant Locators" on page 281). The server exports references, but all objects are initially inactive. For every incoming operation, the POA asks the servant locator to select an appropriate servant. The generated servant locator creates a servant for each incoming operation.

A servant locator is ideal for managing a cache of servants from a very large collection of objects in a database. You can replace the `preinvoke` and `postinvoke` methods in the generated locator with code that looks for servants in a database cache, loads them into the cache if required, and deletes old servants when the cache is full.

-strategy default_servant: The server creates a POA for each interface, and defines a default servant for each POA to handle incoming requests. A server that manages requests for many objects that all use the same interface should probably have a POA that maps all these requests to the same default servant. For more information about using default servants, see "Setting a Default Servant" on page 289.

-ns/-nons

Determines how the server exports object references to the application:

-ns: Use the naming service to publish object references. For each interface, the server binds a reference that uses the interface name, in naming context `IT_GenieDemo`. For example, for interface `Demo_Bank`, the genie binds the reference `IT_GenieDemo/BankDemo_Bank`. If you use this option, the naming service and locator daemon must be running when you start the server.

For more information about the naming service, see Chapter 17 on page 421.

-nons: Write stringified object references to a file. For each interface, the server exports a reference to a file named after the interface with the suffix `ref`—for example `BankDemo_Bank.ref`

The default for this option is set in the configuration file through `default.`.

Implementing a Client

The `-client` option generates client source code in `client.java`. For example:

```
> idlgen java_poa_genie.tcl -client bank.idl
```

When you run this client, it performs the following actions for each interface:

1. Reads an object reference from the file generated by the server—for example, `BankDemo_Bank.ref`.
2. If generated with the `-complete` option, for each operation:
   - Calls the operation and passes random values.
   - Prints out the results.
3. Catches raised exceptions and prints an appropriate message.

Generating Build Files

The `-antfile` option generates a `build.xml` file that contains rules to build the server and client applications. The `build.xml` file provides the following targets:

- `build_all`: Deletes class files, IDL compiler generated files and rebuilds everything.
- `clean`: Deletes all class files.
- `clean_all`: Deletes all generated files.
- `runserver`: Runs the server.
- `runclient`: Runs the client.

To build the client and server, enter this command:

```
> itant build_all
```

Controlling Code Completeness

You can control the extent of the code that is generated for each interface through the `-complete` and `-incomplete` options. These options are valid for server, servant, and client code generation.

The default for this option is set in the configuration file through `default.java_genie.want_complete`.

For example, the following commands generate complete servant and client code and incomplete server mainline code:

```
> idlgen java_poa_genie.tcl -servant -complete bankdemo.idl
> idlgen java_poa_genie.tcl -client -complete bankdemo.idl
> idlgen java_poa_genie.tcl -server -incomplete bankdemo.idl
```

Setting the `-complete` option on servant, server, and client components yields a complete application that you can compile and run. The application performs these tasks:

- The client application calls every operation in the server application and passes random values as `in` parameters.
- The server application returns random values for `inout`/`out` parameters and `return` values.
- Client and server print a message for each operation call, which includes the values passed and returned.

Using the `-complete` option lets you quickly produce a demo or proof-of-concept prototype. It also offers useful models for typical coding tasks, showing how to initialize parameters properly, invoke operations, and throw and catch exceptions.

If you are familiar with calling and parameter passing rules and simply want a starting point for your application, you probably want to use the `-incomplete` option. This option produces minimal code, omitting the bodies of operations, attributes, and client-side invocations.

The sections that follow describe, for each application component, the differences between complete and incomplete code generation. All examples assume the following IDL for interface Account:

```
// IDL:
module BankDemo
{
    // Other interfaces and type definitions omitted...
    interface Account
    {
        exception InsufficientFunds {};
        readonly attribute AccountId  account_id;
        readonly attribute CashAmount balance;
        void withdraw(
            in CashAmount amount
        ) raises (InsufficientFunds);

        void
        deposit(
            in CashAmount amount
        );
    };
}
```

Servant Code

Qualifying the `-servant` option with `-incomplete` or `-complete` yields the required source files for each IDL interface. Either option generates the `AccountImpl.java` source file.

Incomplete servant

The `-incomplete` option specifies to generate servant class `AccountImpl`, which implements the `BankDemo::Account` interface. The implementation of each operation and attribute throws a `CORBA::NO_IMPLEMENT` exception.

For example, the following code is generated for the `withdraw()` operation:

```
public void withdraw(float amount)
    throws org.omg.CORBA.SystemException,
    NoPackage.BankDemo.AccountPackage.InsufficientFunds {
{
    throw org.omg.CORBA.NO_IMPLEMENT();
}
```

All essential elements of IDL code are automatically generated, so you can focus on writing the application logic for each IDL operation.

Complete servant

The `-complete` option specifies to generate the file `idlgen.NoPackage.Random`*`ModuleName`*, which provides the functionality required to generate random values for parameter passing. For example, `idlgen.NoPackage.RandomBankDemo` is generated for the `BankDemo` module.

Each *`interface-name`*`Impl` method is fully implemented to print parameter values and, if required, return a value to the client. For example, the following code is generated for the `withdraw()` operation:

```
// ...
boolean isClient = false;
// ...
public void withdraw(float amount)
    throws org.omg.CORBA.SystemException,
    NoPackage.BankDemo.AccountPackage.InsufficientFunds {

    // Diagnostics: print the values of "in" and "inout"
   parameters
    System.out.println("AccountImpl.withdraw(): called
   with...");
    System.out.println("amount = " + amount);

    // Decide if we want to throw back an exception
    switch (idlgen.RandomFuncs.init().randomlong() % 2) {
      case 1: {
       // Declare and initialise the exception
       NoPackage.BankDemo.AccountPackage.InsufficientFunds IT_ex;
        IT_ex =
       idlgen.NoPackage.BankDemo.RandomAccount.InsufficientFunds(
            isClient);
        throw IT_ex;
      }
      default:
         // Don't throw an exception
         break;
   }
   // Diagnostics
   System.out.println ("AccountImpl.withdraw(): returning"
}
```

Client Code

In a completely implemented client, `java_poa_genie.tcl` generates a source file for each interface, *`interface-name`*`Caller.java`. This source file defines contains a wrapper method for each operation in *`interface-name`*, and the generated client program calls each of these methods. For example, the `BankDemo` client program calls the `deposit()` method in `NoPackage.BankDemo.AccountCaller`, which in turn calls `deposit()` on the Account object. Each method assigns random values to the parameters of operations and prints out the values of parameters that they send, and those that are received back as `out` parameters. Utility methods to assign random values to IDL types are generated in the file `idlgen.NoPackage.Random`*`module-name`*.

An incomplete client contains no invocations.

Both complete and incomplete clients catch raised exceptions and print appropriate messages.

For example, the following client code is generated for the `Account::deposit()` operation in `NoPackage.BankDemo.AccountCaller.deposit()`:

```
public static
void deposit(NoPackage.BankDemo.Account IT_obj)
{

    // Diagnostics: announce our intention to invoke the
   operation.

    // Declare parameters for making the remote call.
    float  amount;

    // Initialise "in" and "inout" parameters with random values
    amount =
   idlgen.NoPackage.RandomBankDemo.CashAmount(isClient);

    // Make the call and handle any exceptions that are thrown
    try {
        IT_obj.deposit(
        amount);
    }
    catch(org.omg.CORBA.SystemException se) {
        System.out.println(
            "deposit failed with the following exception");
        se.printStackTrace(System.out);
        return;
    }
    catch(Exception ex) {
        System.out.println(
            "deposit failed with the following exception");
        ex.printStackTrace(System.out);
        return;
    }

    // If we get this far then no exception was thrown.
    // Success depends on us having gotten back expected
    // values

    System.out.println("deposit done" );
   }
}
```

General Options

You can supply switches that control `java_poa_genie.tcl` genie output:

-jP: By default, `java_poa_genie.tcl` writes all generated application files to a package whose name is initially set in the configuration file through `default.java_genie.package_name`. The distributed configuration file initially sets the package name to `NoPackage`. You can override the default through the -jP switch. For example, the following command puts all generated files in package `my_package`:

```
> idlgen java_poa_genie.tcl -all -jP my_package bank.idl
```

-dir: By default, `java_poa_genie.tcl` writes all output files to the current directory. With the `-dir` option, you can explicitly specify where to generate file output.

-v/-s: By default, `java_poa_genie.tcl` runs in verbose (`-v`) mode. With the `-s` option, you can silence all messaging.

Compiling the Application

To compile a genie-generated application, Orbix must be properly installed on the client and server hosts:

1. Build the application using the `build.xml` file.
2. In separate windows, run first the server, then the client applications.

Configuration Settings

The configuration file `jart_idlgen.cfg` contains default settings for the Java genie `java_poa_genie.tcl` at the scope `default.java_genie`.

Some other settings are not specific to `java_poa_genie.tcl` but are used by the `std/cpp_poa_boa_lib.tcl` library, which maps IDL constructs to their Java equivalents. `java_poa_genie.tcl` uses this library extensively, so these settings affect the output that it generates. They are held in the scope `default.java`.

For a full listing of these settings, refer to the *CORBA Code Generation Toolkit Guide*.

CHAPTER 6

ORB Initialization and Shutdown

The mechanisms for initializing and shutting down the ORB on a client and a server are the same.

Overview

The `main()` of both sever and client must perform these steps:

- Initialize the ORB by calling `org.omg.CORBA.ORB.init()`.
- Shut down and destroy the ORB, by calling `shutdown()` and `destroy()` on the ORB.

In this chapter

This chapter contains the following sections:

Initializing the ORB Runtime

Overview

Before an application can start any CORBA-related activity, it must initialize the ORB runtime by calling `org.omg.CORBA.ORB.init()`. `ORB.init()` returns an object reference to the ORB object; this, in turn, lets the client obtain references to other CORBA objects, and make other CORBA-related calls.

Calling within main()

It is common practice to set a global variable with the ORB reference, so the ORB object is accessible to most parts of the code. However, you should call `ORB.init()` only after you call `main()` to ensure access to command line arguments.

Supplying an ORB name

You can supply an ORB name as an argument; this name determines the configuration information that the ORB uses. If you supply null, Orbix uses the ORB identifier as the default ORB name. ORB names and configuration are discussed in the *Application Server Platform Administrator's Guide*.

Java mapping

The Java mapping provides three forms of initialization:

- Application initialization
- Applet initialization
- Default initialization

These are defined as follows:

```
package org.omg.CORBA;
abstract public class ORB {

    // ...
    public static ORB init(
         String[] args,
         java.util.Properties props);
    public static ORB init(
         java.applet.Applet app,
         java.util.Properties props);
    public static ORB init();
// ...
```

Application initialization

The application initialization method is used with a stand-alone Java application, and returns a new fully functional ORB Java object with each call. This method is defined with two parameters:

- Command arguments as an array of strings.
- A list of Java properties.

Either parameter can be null.

Applet initialization

The applet initialization method is called from an applet, and returns a new fully functional ORB Java object with each call. This method is defined with two parameters:

- The applet.
- A list of Java properties.

Either parameter can be null.

Default initialization

The default initialization method returns a singleton ORB. If called multiple times, it always returns the same Java object. The ORB that this version returns has restricted capabilities. Only the following ORB methods can be invoked on a singleton ORB:

`create_`*`typecode-type`*`_tc()`
`get_primitive_tc()`
`create_any()`

This version of `ORB.init()` primarily serves these purposes:

- Provide a factory for type codes for use by helper classes implementing the `type()` method.
- Create `Any` instances that are used to describe union labels as part of creating a union `TypeCode`.

Registering portable interceptors

During ORB initialization, portable interceptors are instantiated and registered through an ORB initializer. The client and server applications must register the ORB initializer before calling `ORB_init()`. For more information, see "Registering Portable Interceptors" on page 562.

Shutting Down the ORB

Overview

For maximum portability and to ensure against resource leaks, a client or server must always shut down and destroy the ORB at the end of `main()`:

- `shutdown()` stops all server processing, deactivates all POA managers, destroys all POAs, and causes the `run()` loop to terminate. `shutdown()` takes a single Boolean argument; if set to true, the call blocks until the shutdown process completes before it returns control to the caller. If set to false, a background thread is created to handle shutdown, and the call returns immediately.
- `destroy()` destroys the ORB object and reclaims all resources associated with it.

In this section

This section discusses the following topics:

Shutting Down a Client

A client is a CORBA application that does not call org.omg.CORBA.ORB.run() and does not process incoming CORBA invocations.

Example 12 shows how a client is shut down:

Example 12: *Shutting down a CORBA client*

```
// Java
void main(String args[])
{
  org.omg.CORBA.ORB orb;

  //ORB initialization not shown
  ...
  ...
1 orb.shutdown(true);
2 orb.destroy();
}
```

1. A client calls `shutdown()` with the argument `1`(TRUE), causing the `shutdown()` operation to remain blocked until ORB shutdown is complete.
2. The last thing the client does is to call `destroy()`. You are required to call `destroy()` for full CORBA compliancy.

> **Note:** The `destroy()` function has no effect in Orbix. Hence, it can be omitted without affecting the runtime behavior of an Orbix application.

Shutting down a server

Because servers typically process invocations by calling `org.omg.CORBA.ORB.run()`, which blocks indefinitely, `org.omg.CORBA.ORB.shutdown()` cannot be called from the main thread. The following are the main ways of shutting down a server:

- Call `shutdown(0)` from a subthread.
- Call `shutdown(0)` in the context of an operation invocation.

CHAPTER 7

Using Policies

Orbix supports a number of CORBA and proprietary policies that control the behavior of application components.

Most policies are locality-constrained; that is, they apply only to the server or client on which they are set. Therefore, policies can generally be divided into server-side and client-side policies:

- Server-side policies generally apply to the processing of requests on object implementations. Server-side policies can be set programmatically and in the configuration, and applied to the server's ORB and its POAs.
- client-side policies apply to invocations that are made from the client process on an object reference. Client-side policies can be set programmatically and in the configuration, and applied to the client's ORB, to a thread, and to an object reference.

The procedure for setting policies programmatically is the same for both client and server:

1. Create the `CORBA::Policy` object for the desired policy.
2. Add the `Policy` object to a `PolicyList`.
3. Apply the `PolicyList` to the appropriate target—ORB, POA, thread, or object reference.

In this chapter

This chapter discusses issues that are common to all client and server policies.

For detailed information about specific policies, refer to the chapters that cover client and POA development: "Developing a Client" on page 233, and "Managing Server Objects" on page 191.

Creating Policy and PolicyList Objects

Two methods are generally available to create policy objects:

- To apply policies to a POA, use the appropriate policy factory from the `PortableServer::POA` interface.
- Call ORB::create_policy() on the ORB.

After you create the required policy objects, you add them to a `PolicyList`. The `PolicyList` is then applied to the desired application component.

Using POA policy factories

The `PortableServer::POA` interface provides factories for creating `CORBA::Policy` objects that apply only to a POA (see Table 9 on page 198). For example, the following code uses POA factories to create policy objects that specify `PERSISTENT` and `USER_ID` policies for a POA, and adds these policies to a `PolicyList`.

```
import org.omg.CORBA*.;
import org.omg.PortableServer.*;
...

Policy[] policies = new Policy[2];
policies[0]=root_poa.create_lifespan_policy(
    LifespanPolicyValue.PERSISTENT);
policies[1]=root_poa.create_id_assignment_policy(
    IdAssignmentPolicyValue.USER_ID);
```

Orbix also provides several proprietary policies to control POA behavior (see page 163). These policies require you to call `create_policy()` on the ORB to create `Policy` objects, as described in the next section.

Calling create_policy()

You call `create_policy()` on the ORB to create `Policy` objects. For example, the following code creates a `PolicyList` that sets a SyncScope policy of `SYNC_WITH_SERVER`; you can then use this `PolicyList` to set client policy overrides at the ORB, thread, or object scope:

```
import org.omg.Messaging.*;

org.omg.CORBA.Policy[] policies = new org.omg.Policy[1];
org.omg.CORBA.Any policy_value  =
    global_orb.create_any();

SyncScopePolicyValueHelper.insert(
      policy_value,
      SyncScopePolicyValue.SYNC_WITH_SERVER);

policies[0] = orb.create_policy(
    SYNC_SCOPE_POLICY_TYPE.value, policy_value );
```

Setting Orb and Thread Policies

The `CORBA::PolicyManager` interface provides the operations that a program requires to access and set ORB policies. `CORBA::PolicyCurrent` is an empty interface that simply inherits all `PolicyManager` operations; it provides access to client-side policies at the thread scope.

ORB policies override system defaults, while thread policies override policies set on a system or ORB level. You obtain a PolicyManager or PolicyCurrent through `resolve_initial_references()`:

- `resolve_initial_references ("ORBPolicyManager")` returns the ORB's PolicyManager. Both server- and client-side policies can be applied at the ORB level.
- `resolve_initial_references ("PolicyCurrent")` returns a thread's PolicyCurrent. Only client-side policies can be applied to a thread.

The CORBA module contains the following interface definitions and related definitions to manage ORB and thread policies:

```
module CORBA {
    // ...

    enum SetOverrideType
    {
        SET_OVERRIDE,
        ADD_OVERRIDE
    };

    exception InvalidPolicies
    {
        sequence<unsigned short> indices;
    };
```

```
    interface PolicyManager {
        PolicyList
        get_policy_overrides( in PolicyTypeSeq ts );

        void
        set_policy_overrides(
            in PolicyList policies,
            in SetOverrideType set_add
        ) raises (InvalidPolicies);
    };

    interface PolicyCurrent : PolicyManager, Current
    {
    };
    // ...
}
```

set_policy_overrides() overrides policies of the same `PolicyType` that are set at a higher scope. The operation takes two arguments:

- A `PolicyList` sequence of `Policy` object references that specify the policy overrides.
- An argument of type `SetOverrideType`:

  `ADD_OVERRIDE` adds these policies to the policies already in effect.

  `SET_OVERRIDE` removes all previous policy overrides and establishes the specified policies as the only override policies in effect at the given scope.

`set_policy_overrides()` returns a new proxy that has the specified policies in effect; the original proxy remains unchanged.

To remove all overrides, supply an empty `PolicyList` and `SET_OVERRIDE` as arguments.

get_policy_overrides() returns a `PolicyList` of object-level overrides that are in effect for the specified `PolicyTypes`. The operation takes a single argument, a `PolicyTypeSeq` that specifies the `PolicyTypes` to query. If the `PolicyTypeSeq` argument is empty, the operation returns with all overrides for the given scope. If no overrides are in effect for the specified `PolicyTypes`, the operation returns an empty `PolicyList`.

After `get_policy_overrides()` returns a `PolicyList`, you can iterate through the individual `Policy` objects and obtain the actual setting in each one by narrowing it to the appropriate derivation (see "Getting Policies" on page 175).

Setting Server-Side Policies

Orbix provides a set of default policies that are effective if no policy is explicitly set in the configuration or programmatically. You can explicitly set server policies at three scopes, listed in ascending order of precedence:

1. In the configuration, so they apply to all ORBs that are in the scope of a given policy setting. For a complete list of policies that you can set in the configuration, refer to the *Application Server Platform Administrator's Guide*.
2. On the server's ORB, so they apply to all POAs that derive from that ORB's root POA. The ORB has a PolicyManager with operations that let you access and set policies on the server ORB (see "Setting Orb and Thread Policies" on page 165).
3. On individual POAs, so they apply only to requests that are processed by that POA. Each POA can have its own set of policies (see "Using POA Policies" on page 202).

You can set policies in any combination at all scopes. If settings are found for the same policy type at more than one scope, the policy at the lowest scope prevails.

Most server-side policies are POA-specific. POA policies are typically attached to a POA when it is created, by supplying a PolicyList object as an argument to `create_POA()`. The following code creates POA `persistentPOA` as a child of the root POA, and attaches a PolicyList to it:

```
//get an object reference to the root POA
Object poa_obj =
    global_orb.resolve_initial_references("RootPOA");

POA root_poa = POAHelper.narrow(poa_obj);

Policy[] policies=new Policy[2];
policies[0]=root_poa.create_lifespan_policy(
    LifespanPolicyValue.PERSISTENT);
policies[1]=root_poa.create_id_assignment_policy(
    IdAssignmentPolicyValue.USER_ID);

//create a POA for persistent objects
POA persistent_poa = root_poa.create_POA("persistentPOA",
                         poa_manager,
                         policies);
```

In general, you use different sets of policies in order to differentiate among various POAs within the same server process, where each POA is defined in a way that best accommodates the needs of the objects that it processes. So, a server process that contains the POA `persistentPOA` might also contain a POA that supports only transient object references, and only handles requests for callback objects.

For more information about using POA policies, see page 202.

Setting Client Policies

Orbix provides a set of default policies that are effective if no policy is explicitly set in the configuration or programmatically. Client policies can be set at four scopes, listed here in ascending order of precedence:

1. In the configuration, so they apply to all ORBs that are in the scope of a given policy setting. For a complete list of policies that you can set in the configuration, refer to the *Application Server Platform Administrator's Guide*.
2. On the client's ORB, so they apply to all invocations. The ORB has a PolicyManager with operations that let you access and set policies on the client ORB (see "Setting Orb and Thread Policies" on page 165).
3. On a given thread, so they apply only to invocations on that thread. Each client thread has a PolicyCurrent with operations that let you access and set policies on that thread (see page 165).
4. On individual object references, so they apply only to invocations on those objects. Each object reference can have its own set of policies; the Object interface provides operations that let you access and set an object reference's quality of service policies (see "Managing Object Reference Policies" on page 172).

Setting Policies at Different Scopes

You can set policies in any combination at all scopes. If settings are found for the same policy type at more than one scope, the policy at the lowest scope prevails.

For example, the SyncScope policy type determines how quickly a client resumes processing after sending one-way requests. The default SyncScope policy is SYNC_NONE: Orbix clients resume processing immediately after sending one-way requests.

You can set this policy differently on the client's ORB, threads, and individual object references. For example, you might leave the default SyncScope policy unchanged at the ORB scope, set a thread to SYNC_WITH_SERVER; and set certain objects within that thread to SYNC_WITH_TARGET. Given these quality of service settings, the client blocks on one-way invocations as follows:

- Outside the thread, the client never blocks.
- Within the thread, the client always blocks until it knows whether the invocations reached the server.
- For all objects within the thread that have SYNC_WITH_TARGET policies, the client blocks until the request is fully processed.

Managing Object Reference Policies

The CORBA::Object interface contains the following operations to manage object policies:

```
interface Object {
    // ...
    Policy
    get_client_policy(in PolicyType type);

    Policy
    get_policy(in PolicyType type);

    PolicyList
    get_policy_overrides( in PolicyTypeSeq ts );

    Object
    set_policy_overrides(
        in PolicyList policies,
        in SetOverrideType set_add
    ) raises (InvalidPolicies);

    boolean
    validate_connection(out PolicyList inconsistent_policies);
};
```

Note: These operations will be supported in the future IDL-to-Java mapping. In the interim, Orbix supports these operations with helper class com.iona.corba.util.ObjectHelper.

get_client_policy() returns the policy override that is in effect for the specified PolicyType. This method obtains the effective policy override by checking each scope until it finds a policy setting: first at object scope, then thread scope, and finally ORB scope. If no override is set at any scope, the system default is returned.

get_policy() returns the object's effective policy for the specified PolicyType. The effective policy is the intersection of values allowed by the object's effective override —as returned by get_client_policy()—and the policy that is set in the object's IOR. If the intersection is empty, the method

raises exception INV_POLICY. Otherwise, it returns a policy whose value is legally within the intersection. If the IOR has no policy set for the PolicyType, the method returns the object-level override.

get_policy_overrides() returns a PolicyList of overrides that are in effect for the specified PolicyTypes. The operation takes a single argument, a PolicyTypeSeq that specifies the PolicyTypes to query. If the PolicyTypeSeq argument is empty, the operation returns with all overrides for the given scope. If no overrides are in effect for the specified PolicyTypes, the operation returns an empty PolicyList.

After get_policy_overrides() returns a PolicyList, you can iterate through the individual Policy objects and obtain the actual setting in each one by narrowing it to the appropriate derivation (see "Getting Policies" on page 175).

set_policy_overrides() overrides policies of the same PolicyType that are set at a higher scope, and applies them to the new object reference that it returns. The operation takes two arguments:

- A PolicyList sequence of Policy object references that specify the policy overrides.
- An argument of type SetOverrideType:
  - ADD_OVERRIDE adds these policies to the policies already in effect.
  - SET_OVERRIDE removes all previous policy overrides and establishes the specified policies as the only override policies in effect at the given scope.

To remove all overrides, supply an empty PolicyList and SET_OVERRIDE as arguments.

validate_connection() returns true if the object's effective policies allow invocations on that object. This method forces rebinding if one of these conditions is true:

- The object reference is not yet bound.
- The object reference is bound but the current policy overrides have changed since the last binding occurred; or the binding is invalid for some other reason.

The method returns false if the object's effective policies cause invocations to raise the exception `INV_POLICY`. If the current effective policies are incompatible, the output parameter `inconsistent_policies` returns with a `PolicyList` of those policies that are at fault.

If binding fails for a reason that is unrelated to policies, `validate_connections()` raises the appropriate system exception.

A client typically calls `validate_connections()` when its `RebindPolicy` is set to `NO_REBIND`.

Getting Policies

As shown earlier, CORBA::PolicyManager, CORBA::PolicyCurrent, and CORBA::Object each provide operations that allow programmatic access to the effective policies for an ORB, thread, and object. Accessor operations obtain a PolicyList for the given scope. After you get a PolicyList, you can iterate over its Policy objects. Each Policy object has an accessor method that identifies its PolicyType. You can then use the Policy object's PolicyType to narrow to the appropriate type-specific Policy derivation—for example, a SyncScopePolicy object. Each derived object provides its own accessor method that obtains the policy in effect for that scope.

The Messaging module provides these PolicyType definitions:

```
module Messaging
{
    // Messaging Quality of Service

    typedef short RebindMode;

    const RebindMode TRANSPARENT = 0;
    const RebindMode NO_REBIND = 1;
    const RebindMode NO_RECONNECT = 2;

    typedef short SyncScope;

    const SyncScope SYNC_NONE = 0;
    const SyncScope SYNC_WITH_TRANSPORT = 1;
    const SyncScope SYNC_WITH_SERVER = 2;
    const SyncScope SYNC_WITH_TARGET = 3;

    // PolicyType constants

    const CORBA::PolicyType REBIND_POLICY_TYPE              = 23;
    const CORBA::PolicyType SYNC_SCOPE_POLICY_TYPE          = 24;

    // Locally-Constrained Policy Objects

    // Rebind Policy (default = TRANSPARENT)
           readonly attribute RebindMode rebind_mode;
    };
```

```
    interface RebindPolicy : CORBA::Policy {
    // Synchronization Policy (default = SYNC_WITH_TRANSPORT)

    interface SyncScopePolicy : CORBA::Policy {
            readonly attribute SyncScope synchronization;
    };
    ...
}
```

For example, the following code gets the ORB's SyncScope policy:

```
import org.omg.Messaging.*;

// ...
// get reference to PolicyManager

org.omg.CORBA.Object object;
object = orb.resolve_initial_references("ORBPolicyManager");

// narrow
org.omg.CORBA.PolicyManager policy_mgr =
    org.omg.CORBA.PolicyManagerHelper.narrow(object);

// set SyncScope policy at ORB scope (not shown)
// ...

// get SyncScope policy at ORB scope

org.omg.CORBA.Policy[] types = new org.omg.CORBA.Policy[1];
types[0] = SYNC_SCOPE_POLICY_TYPE;

// get PolicyList from ORB's PolicyManager
org.omg.CORBA.Policy[] pList =
    policy_mgr.get_policy_overrides( types );

// evaluate first Policy in PolicyList
org.omg.Messaging.SyncScopePolicy sync_p =
    org.omg.Messaging.SyncScopePolicyHelper.narrow( pList[0] );

org.omg.Messaging.SyncScope sync_policy =
    sync_p.synchronization();

System.out.println(
    "Effective SyncScope policy at ORB level is " + sync_policy;
```

CHAPTER 8

Developing a Server

This chapter explains how to develop a server that implements servants for CORBA objects.

Server tasks

A CORBA server performs these tasks:

- Uses a POA to map CORBA objects to servants, and to process client requests on those objects.
- Implements CORBA objects as POA servants.
- Creates and exports object references for these servants.
- Initializes and shuts down the runtime ORB.
- Passes parameters to server-side operations.

For an overview of server code requirements, see "Enhancing Server Functionality" on page 69. Although throwing exceptions is an important aspect of server programming, it is covered separately in Chapter 12.

For information on ORB initialization and shutdown, see "ORB Initialization and Shutdown" on page 155.

In this chapter

This chapter contains the following sections:

POAs, Skeletons, and Servants

CORBA objects exist in server applications. Objects are implemented, or *incarnated*, by language-specific *servants*. Objects and their servants are connected by the portable object adapter (POA). The POA provides the server-side runtime support that connects server application code to the networking layer of the ORB.

POA tasks

A POA has these responsibilities:

- Create and destroy object references.
- Convert client requests into appropriate calls to application code.
- Synchronize access to objects.
- Cleanly start up and shut down applications.

For detailed information about the POA, see Chapter 8.

POA skeleton class

For each IDL interface, the IDL compiler generates a an abstract POA skeleton class that you compile into the server application. Skeleton classes are abstract classes. You implement skeleton classes in the server application code with servant classes, which define the behavior of the methods that they inherit. Through a servant's inherited connection to a skeleton class, ORB runtime connects that servant back to the CORBA object that it incarnates.

TIE class

The IDL compiler also generates a TIE class, which lets you implement CORBA objects with classes that are unrelated (by inheritance) to skeleton classes. Given Java's restriction on multiple inheritance, TIE class implementations are especially useful for objects that inherit from multiple IDL interfaces. For more information, see "Delegating Servant Implementations" on page 188.

Server request handling

Figure 14 shows how a CORBA server handles an incoming client request, and the stages by which it dispatches that request to the appropriate servant. The server's ORB runtime directs an incoming request to the POA where the object was created. Depending on the POA's state, the request is

either processed or blocked. A POA manager can block requests by rejecting them outright and raising an exception in the client, or by queueing them for later processing.

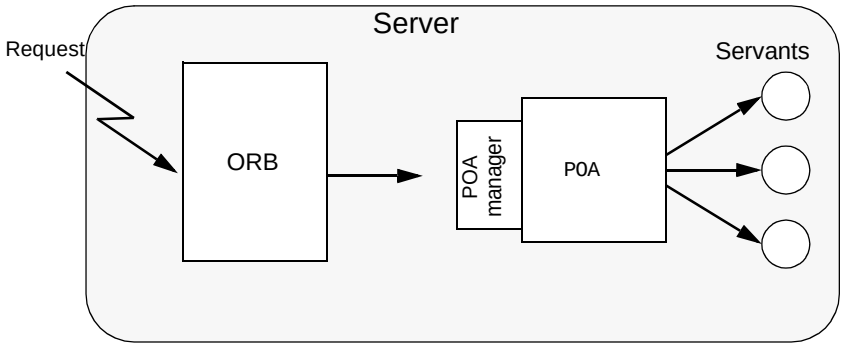


Figure 14: *The server-side ORB conveys client requests to the POA via its manager, and the POA dispatches the request to the appropriate servant.*

Mapping Interfaces to Skeleton Classes

When the ORB receives a request on a CORBA object, the POA maps that request to an instance of the corresponding servant class and invokes the appropriate method.

For example, interface `Account` is defined as follows:

```
module BankDemo
{
    typedef float CashAmount;  // type represents cash
    typedef string AccountId;  // Type represents account IDs
     // ...
    interface Account
    {
        exception InsufficientFunds {};

        readonly attribute AccountId  account_id;
        readonly attribute CashAmount balance;

        void
        withdraw(in CashAmount amount)
        raises (InsufficientFunds);

        void
        deposit( in CashAmount amount);
        };
```

The IDL compiler maps the `Account` interface to the abstract skeleton class `AccountPOA`:

```
package BankDemo;

abstract public class AccountPOA
    extends org.omg.PortableServer.Servant
    implements org.omg.CORBA.portable.InvokeHandler,
               AccountOperations {
    // ...
};
```

The following points are worth noting about the skeleton class:

- `AccountPOA` inherits from `org.omg.PortableServer.Servant`. All skeleton classes inherit from the `Servant` class for two reasons:

  - `Servant` provides functionality that is common to all servants.
  - Servants can be passed generically—you can pass a servant for any type of object as a pointer or reference to `Servant`.
- The skeleton class defines methods that correspond to the interface operations and attributes.
- Because a skeleton class is defined as abstract, you cannot instantiate it. Instead, you must define a concrete servant subclass that implements the skeleton class methods.
- Both the skeleton class and the client stub class implement the same abstract methods that are defined in interface `AccountOperations`.

  Identical signatures preserve location transparency. If the server and client are collocated, the proxy can delegate calls directly to the skeleton without translating or copying data. It also simplifies client and server application development in that one set of parameter passing rules apply to both.

Creating a Servant Class

Each servant class inherits from a skeleton class. The following code defines servant class `AccountImpl`, which extends skeleton class `AccountPOA`. Unlike the skeleton class, the `AccountImpl` class is not abstract, so the server can instantiate `AccountImpl` as a servant.

```
package BankDemo;

import org.omg.CORBA.*;
import org.omg.CORBA.portable.*;
import java.io.*;

import BankDemo.AccountPackage.*;
import BankDemo.*;

public class AccountImpl extends AccountPOA {

    public AccountImpl(java.lang.String account_id,
        AccountDatabase account_db)
    {
        m_account_db = account_db;
        m_account_id = account_id;
        m_balance = m_account_db.read_account(m_account_id);
    }

    protected void finalize() {
        m_account_db.write_account(m_account_id, m_balance);
    }

    protected void save_all() {
        m_account_db.write_account(m_account_id, m_balance);
    }

    public void withdraw(float amount) throws InsufficientFunds {
        if (amount > m_balance) {
            throw new InsufficientFunds();
        }
        m_balance -= amount;
    }
```

```
    public void deposit(float amount) {
        m_balance += amount;
    }

    public String account_id() {
        return m_account_id;
    }

    public float balance() {
        return m_balance;
    }

    private String m_account_id;
    private float m_balance;
    private AccountDatabase m_account_db;
}
```

Note: The choice of name for servant classes is purely a matter of convention. The examples here and elsewhere apply the `Impl` suffix to the original interface name, as in `AccountImpl`. It is always good practice to have a naming convention and use it consistently in your code.

Activating CORBA Objects

In order to enable clients to invoke on CORBA operations, a server must create and export object references. These object references must point back to a CORBA object that is active through its incarnation by a C++ or Java servant.

Activation of a CORBA object is a two-step process:

1. Instantiate the CORBA object's servant. Instantiating a servant does not by itself activate the CORBA object. The ORB runtime remains unaware of the existence of the servant and the corresponding CORBA object.
2. Register the servant and the object's ID in a POA.

this()

The simplest way to activate a CORBA object is by calling `_this()` on the servant. The IDL compiler generates a `_this()` method for each servant skeleton class. `_this()` performs two separate tasks:

- Checks the POA to determine whether the servant is registered with an existing object. If not, `_this()` creates an object from the servant's interface, registers a unique ID for this object in the POA's active object map, and maps this object ID to the servant's address.
- Generates and returns an object reference that includes the object's ID and POA identifier.

In other words, the object is implicitly activated in order to return an object reference.

servant_to_reference()

You can also implicitly activate an object by calling `servant_to_reference()` on the desired POA. This requires you to narrow to the appropriate object; however, there can be no ambiguity concerning the POA in which the object is active, as can happen through using `_this()` (see page 214).

Explicit activation methods

Alternatively, you can explicitly activate a CORBA object: call `activate_object()` or `activate_object_with_id()` on the POA. You can then obtain an object reference by calling `_this()` on the servant. Because the servant is already registered in the POA with an object ID, the method simply returns an object reference.

The ability to activate an object implicitly or explicitly depends on a POA's activation policy. For more information on this topic, see "Using POA Policies" on page 202.

Note: The object reference returned by `_this()` is independent of the servant itself; you must eventually call `release()` on the object. Releasing the object reference has no effect on the corresponding servant.

Handling Output Parameters

Server-side rules

Server-side rules for passing output (in/inout) parameters to the client complement client-side rules. For example, the following IDL defines operation create_account() with two out parameters:

```
module BankDemo {
    // ...

    // Forward declaration of Account
    interface Account;

    interface Bank {
        void create_account(
            (in string name, out Account acct, out string acc_id)
        // ...
    }
    // ...
}
```

Implementation example

The servant that implements this operation must use holder classes for the two out parameters:

```
// in servant class BankImpl
public void create_account( java.lang.String name,
                AccountHolder acct, StringHolder acc_id ) {

    AccountImpl new_acct =
        new AccountImpl (account_id, account_db);

    // set AccountHolder value to Account object reference
    acct.value = _this(new_acct);
    // ...
}
```

For more information about holder classes, see “Passing Parameters in Client Invocations” on page 249.

Delegating Servant Implementations

Previous examples show how Orbix uses inheritance to associate servant classes and their implementations with IDL interfaces. By inheriting from IDL-derived skeleton classes, servants establish their connection to the corresponding IDL interfaces, and thereby make themselves available to client requests.

Alternatively, you can explicitly associate, or *tie* a servant and its operations to the appropriate IDL interface through tie classes. The tie approach lets you implement CORBA objects with classes that are unrelated (by inheritance) to skeleton classes.

The TIE approach is especially useful when implementing CORBA objects whose IDL definitions inherit from multiple interfaces. Given Java single-inheritance restrictions on classes, a servant class that inherits from an abstract POA skeleton class cannot inherit from any other class. Therefore, it must implement all the methods that the skeleton class defines; it cannot reuse methods from other classes. By contrast, a tie servant is free to inherit from any class.

Creating tie-based servants

Tie-based servants rely on two components:

- A *tie object* implements the CORBA object; however, unlike the inherited approach, the class that it instantiates does not inherit from any of the IDL-generated base skeleton classes.
- A *tie servant* instantiates a tie class, which the IDL compiler generates when you run it with the `-xTIE` switch. The POA regards a tie servant as the actual servant of an object. Thus, all POA operations on a servant such as `activate_object()` take the tie servant as an argument. The tie servant receives client invocations and forwards them to the tie object.

To create a tie servant and associate it with a tie object:

1. Instantiate the tie object
2. Create the tie object through the tie class constructor:

```
tie-servant = tie-constructor(tied-object, poa);
```

Example

For example, given an IDL specification that includes interface `BankDemo::Bank`, the IDL compiler can generate tie class `BankDemo.BankPOATie`. This class supplies a number of operations that enable its tie servant to control the tie object.

Given implementation class `BankImpl`, you can instantiate a tie object and create tie servant `bank_srv_tie` for it as follows:

```
// instantiate tie object and create its tie servant
BankImpl tie_object = new BankImpl();
BankDemo.BankPOATie bank_srv_tie =
    new BankDemo.BankPOATie(tie_object, the_poa);
```

Given this tie servant, you can use it to create an object reference:

```
//create an object reference for bank servant
Bank bankref = bank_srv_tie._this();
```

When the POA receives client invocations on the `bankref` object, it relays them to tie servant `bank_srv_tie`, which delegates them to the bank tie object for processing.

Removing tie objects and servants

You remove a tie servant from memory like any other servant—for example, with `org.omg.PortableServer.POA.deactivate_object()`. If the tie servant's tie object implements only a single object, the tie object is also removed.

Explicit Event Handling

When you call `ORB::run()`, the ORB gets the thread of control to dispatch events. This is acceptable for a server that only processes CORBA requests. However, if your process must also support a GUI or uses another networking stack, you also must be able to monitor incoming events that are not CORBA client requests.

The `ORB` interface methods `work_pending()` and `perform_work()` let you poll the ORB's event loop for incoming requests:

- `work_pending()` returns true if the ORB's event loop has at least one request ready to process.
- `perform_work()` processes one or more requests before it completes and returns the thread of control to the application code. The amount of work processed by this call depends on the threading policies and the number of queued requests; however, `perform_work()` guarantees to return periodically so you can handle events from other sources.

CHAPTER 9

Managing Server Objects

A portable object adapter, or POA, maps CORBA objects to language-specific implementations, or servants, in a server process. All interaction with server objects takes place via the POA.

A POA identifies objects through their object IDs, which are encapsulated within the object requests that it receives. Orbix views an object as *active* when its object ID is mapped to a servant; the servant is viewed as *incarnating* that object. By abstracting an object's identity from its implementation, a POA enables a server to be portable among different implementations.

In this chapter

This chapter shows how to create and manage a POA within a server process, covering the following topics:

Mapping Objects to Servants

Figure 15 shows how a POA manages the relationship between CORBA objects and servants, within the context of a client request. A client references an object or invokes a request on it through an interoperable object reference (*IOR*). This IOR encapsulates the information required to find the object, including its server address, POA, and object ID—in this case, *A*. On receiving the request, the POA uses the object's ID to find its servant. It then dispatches the requested operation to the servant via the server skeleton code, which extracts the operation's parameters and passes the operation as a language-specific call to the servant.

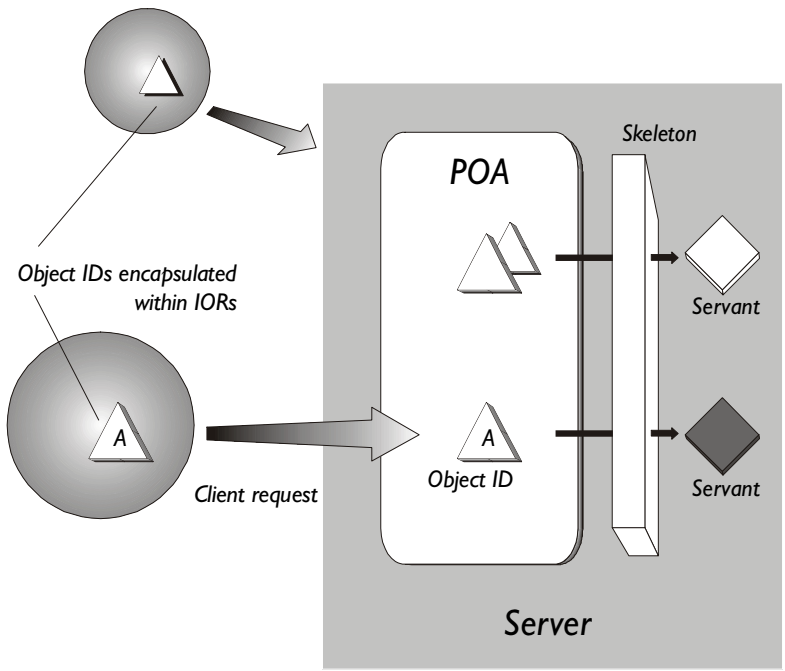


Figure 15: *A portable object adapter (POA) maps abstract objects to their concrete implementations (servants)*

Depending on a POA's policies, a servant can be allowed to incarnate only one object; or it can incarnate multiple objects. During an object's lifetime, it can be activated multiple times by successive servant incarnations.

Mapping options

A POA can map between objects and servants in several ways:

- An *active object map* retains object-servant mappings throughout the lifetime of its POA, or until an object is explicitly deactivated. Before a POA is activated, it can anticipate incoming requests by mapping known objects to servants, and thus facilitate request processing.
- A *servant manager* maps objects to servants on demand, either on the initial object request, or on every request. Servant managers can enhance control over servant instantiation, and help avoid or reduce the overhead incurred by a static object-servant mapping.
- A single *default servant* can be used to handle all object requests. A POA that uses a default servant incurs the same overhead no matter how many objects it processes.

Depending on its policies, a POA can use just one object-mapping method, or several methods in combination. For more information, see "Enabling the Active Object Map" on page 203.

Creating a POA

All server processes in a location domain use the same root POA, which you obtain by calling `resolve_initial_references("POA")`. The root POA has predefined policies which cannot be changed (see page 201). Within each server process, the root POA can spawn one or more child POAs. Each child POA provides a unique namespace; and each can have its own set of policies, which determine how the POA implements and manages object-servant mapping. Further, each POA can have its own POA manager and servant manager.

Using multiple POAs

A number of objectives can justify the use of multiple POAs within the same server. These include:

- *Partition the server into logical or functional groups of servants.* You can associate each group with a POA whose policies conform with the group's requirements. For example, a server that manages Customer and Account servants can provide a different POA for each set of servants.

  You can also group servants according to common processing requirements. For example, a POA can be configured to generate object references that are valid only during the lifespan of that POA, or across all instantiations of that POA and its server. POAs thus offer built-in support for differentiating between persistent and transient objects.
- *Independently control request processing for sets of objects.* A POA manager's state determines whether a POA is active or inactive; it also determines whether an active POA accepts incoming requests for processing, or defers them to a queue (see "Processing Object Requests" on page 204). By associating POAs with different managers, you can gain finer control over object request flow.
- *Choose the method of object-servant binding that best serves a given POA.* For example, a POA that processes many objects can map all of them to the same default servant, incurring the same overhead no matter how many objects it processes.

Procedure for creating a POA

Creating a POA consists of these steps:

1. Set the POA policies.

   Before you create a POA, establish its desired behavior through a CORBA PolicyList, which you attach to the new POA on its creation. Any policies that are explicitly set override a new POA's default policies (refer to Table 9 on page 198).

2. Create the POA by calling `create_POA()` on an existing POA.
3. If the POA has a policy of `USE_SERVANT_MANAGER`, register its servant manager by calling `set_servant_manager()` on the POA.
4. Enable the POA to receive client requests by calling `activate()` on its POA manager.

Setting POA Policies

A new POA's policies are set when it is created. You can explicitly set a POA's policies through a CORBA PolicyList object, which is a sequence of Policy objects.

Java applications represent a PolicyList object as an array of Policy objects.

Creating Policy objects

The `PortableServer::POA` interface provides factories to create CORBA Policy object types (see Table 9 on page 198). If a Policy object type is proprietary to Orbix, you must create the Policy object by calling `create_policy()` on the ORB (see "Setting proprietary policies for a POA" on page 199). In all cases, you attach the PolicyList object to the new POA. All policies that are not explicitly set in the PolicyList are set to their defaults.

For example, the following code creates policy objects of `PERSISTENT` and `USER_ID`:

```
import org.omg.CORBA*.;
import org.omg.PortableServer.*;
...

Policy[] policies = new Policy[2];
policies[0]=root_poa.create_lifespan_policy(
    LifespanPolicyValue.PERSISTENT);
policies[1]=root_poa.create_id_assignment_policy(
    IdAssignmentPolicyValue.USER_ID);
```

With the `PERSISTENT` policy, a POA can create object references that remain valid across successive instantiations of this POA and its server process. The `USER_ID` policy requires the application to autoassign all object IDs for a POA.

Attaching policies to a POA

After you create a PolicyList object, you attach it to a new POA by supplying it as an argument to `create_POA()`. The following code creates POA `persistentPOA` as a child of the root POA, and attaches to it the PolicyList object just shown:

```
//get an object reference to the root POA
Object poa_obj =
    global_orb.resolve_initial_references("RootPOA");

POA root_poa =
    POAHelper.narrow(poa_obj);

Policy[] policies=new Policy[2];
policies[0]=root_poa.create_lifespan_policy(
    LifespanPolicyValue.PERSISTENT);
policies[1]=root_poa.create_id_assignment_policy(
    IdAssignmentPolicyValue.USER_ID);

//create a POA for persistent objects
POA persistent_poa = root_poa.create_POA("persistentPOA",
                         poa_manager,
                         policies);
```

In general, POA policies let you differentiate among various POAs within the same server process, where each POA is defined in a way that best accommodates the needs of the objects that it processes. For example, a server process that contains the POA `persistentPOA` might also contain a POA that supports only transient object references, and only handles requests for callback objects.

POA Policy factories

The `PortableServer::POA` interface contains factory methods for creating CORBA Policy objects:

Table 9: *POA policy factories and argument options*

POA policy factories	Policy options
`create_id_assignment_policy()`	`SYSTEM_ID` *(default)* `USER_ID`
`create_id_uniqueness_policy()`	`UNIQUE_ID` *(default)* `MULTIPLE_ID`

Table 9: *POA policy factories and argument options*

POA policy factories	Policy options
`create_implicit_activation_policy()`	`NO_IMPLICIT_ACTIVATION` *(default)* `IMPLICIT_ACTIVATION`
`create_lifespan_policy()`	`TRANSIENT` *(default)* `PERSISTENT`

`create_request_processing_policy()`	`USE_ACTIVE_OBJECT_MAP_ONLY` *(default)*
	`USE_DEFAULT_SERVANT`
	`USE_SERVANT_MANAGER`

`create_servant_retention_policy()`	`RETAIN` *(default)* `NON_RETAIN`
`create_thread_policy()`	`ORB_CTRL_MODEL` *(default)* `SINGLE_THREAD_MODEL`

For specific information about these methods, refer to their descriptions in the *CORBA Programmer's Reference*.

Setting proprietary policies for a POA

Orbix provides several proprietary policies to control POA behavior. To set these policies, call `create_policy()` on the ORB to create Policy objects with the desired policy value, and add these objects to the POA's PolicyList.

For example, Orbix provides policies that determine how a POA handles incoming requests for any object as it undergoes deactivation. You can specify a `DISCARD` policy for a POA so it discards all incoming requests for deactivating objects:

```
import com.iona.corba.*;

org.omg.CORBA.Policy[] policies = new org.omg.CORBA.Policy[1];
org.omg.CORBA.Any obj_deactivation_policy_value =
    global_orb.create_any();
ObjectDeactivationPolicyValueHelper.insert(
    obj_deactivation_policy_value,
    ObjectDeactivationPolicyValue.DISCARD);

policies[0] = global_orb.create_policy(
    ( OBJECT_DEACTIVATION_POLICY_ID.value,
     obj_deactivation_policy_value );
```

Orbix-proprietary policies

You can attach the following Orbix-proprietary Policy objects to a POA's `PolicyList`:

ObjectDeactivationPolicy controls how the POA handles requests that are directed at deactivating objects. This policy is valid only for a POA that uses a servant activator to control object activation. For more information, see "Setting deactivation policies" on page 279.

PersistenceModePolicy can specify a policy of `DIRECT_PERSISTENCE`, so that the POA uses a well-known address in the IORs that it generates for persistent objects. This policy is valid only for a POA that has a `PERSISTENT` lifespan policy. For more information, see "Direct persistence" on page 206.

WellKnownAddressingPolicy sets transport configuration data—for example, address information for persistent objects that use a well-known address, or IIOP buffer sizes. For more information, see "Direct persistence" on page 206.

DispatchWorkQueuePolicy specifies the work queue used to process requests for a POA whose threading policy is set to `ORB_CTRL_MODEL`. All requests for the POA are dispatched in a thread controlled by the specified work queue. For more information, see "Work Queues" on page 220.

WorkQueuePolicy specifies the work queue used by network transports to read requests for the POA. For more information, see "Work Queues" on page 220.

InterdictionPolicy disables the proxification of the POA when using the Iona firewall proxy service. A POA with this policy set to `DISABLE` will never be proxified. For more information, see "Controlling POA Proxification" on page 230.

Root POA Policies

The root POA has the following policy settings, which cannot be changed:

Policy	Default setting
Id Assignment	`SYSTEM_ID`
Id Uniqueness	`UNIQUE_ID`
Implicit Activation	`IMPLICIT_ACTIVATION`
Lifespan	`TRANSIENT`
Request Processing	`USE_ACTIVE_OBJECT_MAP_ONLY`
Servant Retention	`RETAIN`
Thread	`ORB_CTRL_MODEL`

Using POA Policies

Overview

A POA's policies play an important role in determining how the POA implements and manages objects and processes client requests. While the root POA has a set of predefined policies that cannot be changed, any POA that you create can have its policies explicitly set.

In this section

The following sections describe POA policies and setting options:

Enabling the Active Object Map

A POA's servant retention policy determines whether it uses an active object map to maintain servant-object associations. Depending on its request processing policy (see page 204), a POA can rely exclusively on an active object map to map object IDs to servants, or it can use an active object map together with a servant manager and/or default servant. A POA that lacks an active object map must use either a servant manager or a default servant to map between objects and servants.

You specify a POA's servant retention policy by calling `create_servant_retention_policy()` with one of these arguments:

RETAIN: The POA retains active servants in its active object map.

NON_RETAIN: The POA has no active object map. For each request, the POA relies on the servant manager or default servant to map between an object and its servant; all mapping information is destroyed when request processing returns. Thus, a `NON_RETAIN` policy also requires that the POA have a request processing policy of `USE_DEFAULT_SERVANT` or `USE_SERVANT_MANAGER` (see "Processing Object Requests" on page 204).

Servant manager and servant retention policy

If a POA has a policy of `USE_SERVANT_MANAGER`, its servant retention policy determines whether it uses a servant activator or servant locator as its servant manager. A `RETAIN` policy requires the use of a servant activator; a `NON_RETAIN` policy requires the use of a servant locator. For more information about servant managers, see Chapter 11.

Processing Object Requests

A POA's request processing policy determines how it locates a servant for object requests. Four options are available:

- Maintain a permanent map, or *active object map*, between object IDs and servants and rely exclusively on that map to process all object requests.
- Activate servants on demand for object requests.
- Locate a servant for each new object request.
- Map object requests to a single default servant.

For example, if the application processes many lightweight requests for the same object type, the server should probably have a POA that maps all these requests to the same default servant. At the same time, another POA in the same server might be dedicated to a few objects that each use different servants. In this case, requests can probably be processed more efficiently if the POA is enabled for permanent object-servant mapping.

You set a POA's request processing policy by calling `create_request_processing_policy()` and supplying one of these arguments:

- USE_ACTIVE_OBJECT_MAP_ONLY
- USE_SERVANT_MANAGER
- USE_DEFAULT_SERVANT

USE_ACTIVE_OBJECT_MAP_ONLY: All object IDs must be mapped to a servant in the active object map; otherwise, Orbix returns an exception of `OBJECT_NOT_EXIST` to the client.

During POA initialization and anytime thereafter, the active object map is populated with all object-servant mappings that are required during the POA's lifetime. The active object map maintains object-servant mappings until the POA shuts down, or an object is explicitly deactivated through `deactivate_object()`.

Typically, a POA can rely exclusively on an active object map when it processes requests for a small number of objects.

This policy requires POA to have a servant retention policy of `RETAIN`. (see "Enabling the Active Object Map" on page 203).

USE_SERVANT_MANAGER: The POA's servant manager finds a servant for the requested object. Depending on its servant retention policy, the POA can implement one of two servant manager types, either a *servant activator* or a *servant locator*:

- A servant activator can be registered with a POA that has a `RETAIN` policy. The servant activator incarnates servants for inactive objects on receiving an initial request for them. The active object map retains mappings between objects and their servants; it handles all subsequent requests for this object.
- If the POA has a policy of `NON_RETAIN` (the POA has no active object map), a servant locator must find a servant for an object on each request; otherwise, an `OBJ_ADAPTER` exception is returned when clients invoke requests.

`USE_SERVANT_MANAGER` requires the application to register a servant manager with the POA by calling `set_servant_manager()`.

For more information about servant managers, see Chapter 11.

USE_DEFAULT_SERVANT: The POA dispatches requests to the default servant when it cannot otherwise find a servant for the requested object. This can occur because the object's ID is not in the active object map, or the POA's servant retention policy is set to `NON_RETAIN`.

Set this policy for a POA that needs to process many objects that are instantiated from the same class, and thus can be implemented by the same servant.

This policy requires the application to register the POA's default servant by calling `set_servant()` on the POA; it also requires the POA's ID uniqueness policy to be set to `MULTIPLE_ID`, so multiple objects can use the default servant.

Setting Object Lifespan

A POA creates object references through calls to `create_reference()` or `create_reference_with_id()`. The POA's lifespan policy determines whether these object references are persistent—that is, whether they outlive the process in which they were created. A persistent object reference is one that a client can successfully reissue over successive instantiations of the target server and POA.

You specify a POA's lifespan policy by calling `create_lifespan_policy()` with one of these arguments

TRANSIENT: (default policy) Object references do not outlive the POA in which they are created. After a transient object's POA is destroyed, attempts to use this reference yield the exception `CORBA::OBJECT_NOT_EXIST`.

PERSISTENT: Object references can outlive the POA in which they are created.

Transient object references

When a POA creates an object reference, it encapsulates it within an IOR. If the POA has a `TRANSIENT` policy, the IOR contains the server process's current location—its host address and port. Consequently, that object reference is valid only as long as the server process remains alive. If the server process dies, the object reference becomes invalid.

Persistent object references

If the POA has a `PERSISTENT` policy, the IOR contains the address of the location domain's implementation repository, which maps all servers and their POAs to their current locations. Given a request for a persistent object, the location daemon uses the object's "virtual" address first, and looks up the server process's actual location via the implementation repository.

Direct persistence

Occasionally, you might want to generate persistent object references that avoid the overhead of using the location daemon. In this case, Orbix provides the proprietary policy of `DIRECT_PERSISTENCE`. A POA with policies of `PERSISTENT` and `DIRECT_PERSISTENCE` generates IORs that contain a well-known address list for the server process.

A POA that uses direct persistence must also indicate where the configuration sets the well-known address list to be embedded in object references. In order to do this, two requirements apply:

- The configuration must contain a well-known address configuration variable, with this syntax:

  prefix:*transport*:addr_list=[*address-spec [,...]*]

- The POA must have a `WELL_KNOWN_ADDRESSING_POLICY` whose value is set to *`prefix`*.

For example, you might create a well-known address configuration variable in name scope `MyConfigApp` as follows:

```
MyConfigApp {
    ...
    wka:iiop:addr_list=["host.com:1075"];
    ...
}
```

Given this configuration, a POA is created in the ORB `MyConfigApp` can have its PolicyList set so it generates object references that use direct persistence, as follows:

```
import com.iona.corba.*;
import com.iona.IT_CORBA.*;
import com.iona.IT_PortableServer.*;

// Set up IONA policies
org.omg.CORBA.Any persistent_mode_policy_value =
    global_orb.create_any();
org.omg.CORBA.Any well_known_addressing_policy_value =
    global_orb.create_any();

PersistenceModePolicyValueHelper.insert(
    persistent_mode_policy_value,
    PersistenceModePolicyValue.DIRECT_PERSISTENCE);
well_known_addressing_policy_value.insert_string(
    "wka");
```

```
org.omg.CORBA.Policy[] policies=new Policy[] {
    root_poa.create_lifespan_policy(
        LifespanPolicyValue.PERSISTENT),
    root_poa.create_id_assignment_policy(
        IdAssignmentPolicyValue.USER_ID),
    global_orb.create_policy(
        PERSISTENCE_MODE_POLICY_ID.value,
        persistence_mode_policy_value),
    global_orb.create_policy(
        WELL_KNOWN_ADDRESSING_POLICY_ID.value,
        well_known_addressing_policy_value),
};
...
```

Object lifespan and ID assignment

A POA's lifespan and ID assignment policies have dependencies upon one another.

`TRANSIENT` and `SYSTEM_ID` are the default settings for a new POA, becuase system-assigned IDs are sufficient for transient object references. The appication does not need tight control over the POA's ID becuase the POA's object reference is only valid for the POA's current incarnation.

However, `PERSISTENT` and `USER_ID` policies are usually set together, because applications require explicit control over the object IDs of its persistent object references. When using persistent object references the POA's name is part of the information used to resolve an object's IOR. For this reason, there is a possibility of conflicts when using multiple POA's with the same name and a lifespan policy of `PERSISTENT`. This is particularly true when using indirect persistent IORs.

Assigning Object IDs

The ID assignment policy determines whether object IDs are generated by the POA or the application. Specify the POA's ID assignment policy by calling `create_id_assignment_policy()` with one of these arguments:

SYSTEM_ID: The POA generates and assigns IDs to its objects. Typically, a POA with a `SYSTEM_ID` policy manages objects that are active for only a short period of time, and so do not need to outlive their server process. In this case, the POA also has an object lifespan policy of `TRANSIENT`. Note, however, that system-generated IDs in a persistent POA are unique across all instantiations of that POA.

USER_ID: The application assigns object IDs to objects in this POA. The application must ensure that all user-assigned IDs are unique across all instantiations of the same POA.

`USER_ID` is usually assigned to a POA that has an object lifespan policy of `PERSISTENT`—that is, it generates object references whose validity can span multiple instantiations of a POA or server process, so the application requires explicit control over object IDs.

Activating Objects with Dedicated Servants

A POA's ID uniqueness policy determines whether it allows a servant to incarnate more than one object. You specify a POA's ID uniqueness policy by calling `create_id_uniqueness_policy()` with one of these arguments:

UNIQUE_ID: Each servant in the POA can be associated with only one object ID.

MULTIPLE_ID: Any servant in the POA can be associated with multiple object IDs.

Note: If the same servant is used by different POAs, that servant conforms to the uniqueness policy of each POA. Thus, it is possible for the same servant to be associated with multiple objects in one POA, and be restricted to one object in another.

Activating Objects

A POA's activation policy determines whether objects are explicitly or implicitly associated with servants. If a POA is enabled for explicit activation, you activate an object by calling `activate_object()` or `activate_object_with_id()` on the POA. A POA that supports implicit activation allows the server application to call the `_this()` function on a servant to create an active object (see "Implicit Object Activation" on page 214).

The activation policy determines whether the POA supports implicit activation of servants.

Specify the POA's activation policy by supplying one of these arguments:

NO_IMPLICIT_ACTIVATION: (default) The POA only supports explicit activation of servants.

IMPLICIT_ACTIVATION: The POA supports implicit activation of servants. This policy requires that the POA's object ID assignment policy be set to `SYSTEM_ID`, and its servant retention policy be set to `RETAIN`.

For more information, see "Implicit Object Activation" on page 214.

Setting Threading Support

Specify the POA's thread policy by supplying one of these arguments:

ORB_CTRL_MODEL: The ORB is responsible for assigning requests for an ORB-controlled POA to threads. In a multi-threaded environment, concurrent requests can be delivered using multiple threads.

SINGLE_THREAD_MODEL: Requests for a single-threaded POA are processed sequentially. In a multi-threaded environment, all calls by a single-threaded POA to implementation code (servants and servant managers) are made in a manner that is safe for code that does not account for multi-threading.

Multiple single-threaded POAs might need to cooperate to ensure that calls are safe when they share implementation code such as a servant manager.

Default work queues

Orbix maintains for each ORB two default work queues, one manual and the other automatic. Depending on its thread policy, a POA that lacks its own work queue uses one of the default work queues to process requests:

- A POA with a threading policy of `SINGLE_THREAD_MODEL` uses the manual work queue. To remove requests from the manual work queue, you must call either `ORB::perform_work()` or `ORB::run()` within the main thread.
- A POA with a threading policy of `ORB_CTRL_MODEL` uses the automatic work queue. Requests are automatically removed from this work queue; however, because `ORB::run()` blocks until the ORB shuts down, an application can call this method to detect when shutdown is complete.

Both threading policies assume that the ORB and the application are using compatible threading synchronization. All uses of the POA within the server must conform to its threading policy.

For information about creating a POA workqueue, see page 220.

Explicit Object Activation

If the POA has an activation policy of `NO_IMPLICIT_ACTIVATION`, the server must call either `activate_object()` or `activate_object_with_id()` on the POA to activate objects. Either of these calls registers an object in the POA with either a user-supplied or system-generated object ID, and maps that object to the specified servant.

After you explicitly activate an object, you can obtain its object reference in two ways:

- Use the object's ID to call `id_to_reference()` on the POA where the object was activated. `id_to_reference()` uses the object's ID to obtain the information needed to compose an object reference, and returns that reference to the caller.
- Call `_this()` on the servant. Because the servant is already registered in the POA with an object ID, the function composes an object reference from the available information and returns that reference to the caller.

Implicit Object Activation

A server activates an object implicitly by calling `_this()` on the servant designated to incarnate that object. `_this()` is valid only if the POA that maintains these objects has policies of `RETAIN`, `SYSTEM_ID`, and `IMPLICIT_ACTIVATION`; otherwise, it raises a `WrongPolicy` exception. Thus, implicit activation is generally a good option for a POA that maintains a relatively small number of transient objects.

Calling _this()

`_this()` performs two separate tasks:

- Checks the POA to determine whether the servant is registered with an existing object. If it is not, `_this()` creates an object from the servant's interface, registers a new ID for this object in the POA's active object map, and maps this object ID to the servant.
- Generates and returns an object reference.

In other words, the object is implicitly activated in order to return an object reference.

You can call `_this()` on a servant in two ways:

- Within an operation that is invoked on the servant's object.
- Outside an operation.

Calling _this() Inside an Operation

If called inside an operation, `_this()` returns a reference to the object on which the operation was invoked. Thus, a servant can always obtain a reference to the object that it incarnates—for example, in order to register the object as a callback with another object.

The following interface defines the `get_self()` operation, whose implementation returns a reference to the same interface:

```
interface Whatever {
    Whatever get_self();
};
```

You might implement this operation as follows:

```
Whatever get_self() throws org.omg.CORBA.SystemException
{
    return _this();         // Return reference to self
}
```

Calling _this() Outside an Operation

You can activate an object and obtain a reference to it by calling `_this()` on a servant. This object reference must include information that it obtains from the POA in which the object is registered: the fully qualified POA name, protocol information, and the object ID that is registered in the POA's active object map. `_this()` determines which POA to use by calling `_default_POA()` on the servant.

`_default_POA()` is inherited from the `ServantBase` class:

```
public class org.omg.PortableServerServant {
    public POA _default_POA() {}
    // ...
};
```

Servant inheritance of _default_POA() implementation

All skeleton classes and the servants that implement them derive from `Servant`, and therefore inherit its implementation of `_default_POA()`. The inherited `_default_POA()` always returns the root POA. Thus, calling `_this()` on a servant that does not override `_default_POA()` returns a transient object reference that points back to the root POA. All invocations on that object are processed by the root POA.

As seen earlier, an application typically creates its own POAs to manage objects and client requests. For example, to create and export persistent object references, you must create a POA with a `PERSISTENT` lifespan policy and use it to generate the desired object references. If this is the case, you must be sure that the servants that incarnate those objects also override `_default_POA()`; otherwise, calling `_this()` on those servants returns transient object references whose mappings to servants are handled by the root POA.

Note: To avoid ambiguity concerning the POA in which an object is implicitly activated, call `servant_to_reference()` on the desired POA instead of `_this()`. While using `servant_to_reference()` requires you to narrow to the appropriate object, the extra code is worth the extra degree of clarity that you achieve.

Overriding _default_POA()

To ensure that `_this()` uses the right POA to generate object references, an application's servants must override the default POA. You can do this three ways:

Override _default_POA() to throw a system exception. For example, `_default_POA()` can return system exception `CORBA::INTERNAL`. This prevents use of `_this()` to generate any object references for that servant.

By overriding `_default_POA()` to throw an exception, you ensure that attempts to use `_this()` yield an immediate error instead of a subtly incorrect behavior that must be debugged later. Instead, you must create object references with calls to either `create_reference()` or `create_reference_with_id()` (see page 290), then explicitly map objects to servants—for example, through a servant manager, or via the active object map by calling `activate_object_with_id.()`.

Disabling `_default_POA()` also prevents you from calling `_this()` to obtain an existing object reference for a servant. To obtain the reference, you must call `servant_to_reference()`.

Override _default_POA() in each servant to return the correct POA. Calls to `_this()` are guaranteed to use the correct POA. This approach also raises a `WrongPolicy` exception if the POA that you set for a servant has invalid policies for implicit activation. such as `USER_ID`.

This approach requires the application to maintain a reference for the servant's POA. If all servants use the same POA, you can set the reference in a global variable or a static private member. However, if a server uses unique POAs for different groups of servants, each servant must carry the overhead of an additional (non-static) data member.

Override _default_POA() in a common base class. Servant classes that need to override `_default_POA()` can inherit from a common base class that contains an override definition. This approach to overriding `_default_POA()` has two advantages:

- You only need to write the overriding definition of `_default_POA()` once.
- If you define a servant class that inherits from multiple servant classes, you avoid inheriting conflicting definitions of the `_default_POA()` method.

Managing Request Flow

Each POA is associated with a `POAManager` object that determines whether the POA can accept and process object requests. When you create a POA, you specify its manager by supplying it as an argument to `create_POA()`. This manager remains associated with the POA throughout its life span.

`create_POA()` can specify either an existing POA manager, or `null` to create a `POAManager` object. You can obtain the `POAManager` object of a given POA by calling `the_POAManager()` on it. By creating POA managers and using existing ones, you can group POAs under different managers according to their request processing needs. Any POA in the POA hierarchy can be associated with a given manager; the same manager can be used to manage POAs in different branches.

POA manager states

A POA manager can be in four different states. The `POAManager` interface provides four operations to change the state of a POA manager, as shown in Table 10.

Table 10: *POA manager states and interface operations*

State	Operation	Description
Active	`activate()`	Incoming requests are accepted for processing. When a POA manager is created, it is initially in a holding state. Until you call `activate()` on a POA's manager, all requests sent to that POA are queued.
Holding	`hold_requests()`	All incoming requests are queued. If the queue fills to capacity, incoming requests are returned with an exception of `TRANSIENT`.

Table 10: *POA manager states and interface operations*

State	Operation	Description
Discarding	`discard_requests()`	All incoming requests are refused and a system exception of `TRANSIENT` is raised to clients so they can reissue their requests. A POA manager is typically in a discarding state when the application detects that an object or the POA in general cannot keep pace with incoming requests. A POA manager should be in a discarding state only temporarily. On resolution of the problem that required this call, the application should restore the POA manager to its active state with `activate()`.
Inactive	`deactivate()`	The POA manager is shutting down and destroying all POAs that are associated with it. Incoming requests are rejected with the exception `CORBA::OBJ_ADAPTER`.

Holding state

The POA manager of the root POA is initially in a holding state, as is a new POA manager. Until you call `activate()` on a POA's manager, all requests sent to that POA are queued. `activate()` can also reactivate a POA manager that has reverted to a holding state (due to a `hold_requests()` call) or is in a discarding state (due to a `discard_requests()` call).

If a new POA is associated with an existing active POA manager, it is unnecessary to call `activate()`. However, it is generally a good idea to put a POA manager in a holding state before creating a new POA with it.

The queue for a POA manager that is in a holding state has limited capacity, so this state should be maintained for a short time only. Otherwise, the queue is liable to fill to capacity with pending requests. When this happens, all subsequent requests return to the client with a `TRANSIENT` exception.

Work Queues

Overview

Orbix provides two proprietary policies, which allow you to associate a `WorkQueue` with a POA and thereby control the flow of incoming requests for that POA:

DispatchWorkQueuePolicy associates a work queue with an `ORB_CTRL_MODEL` POA. All work items for the POA are processed by the work queue in a thread owned by the work queue.

WorkQueuePolicy associates a work queue with any POA. The specified work queue will be used by the underlying network transports for reading requests from the POA.

Interface

A work queue has the following interface definition:

```
// IDL
interface WorkQueue
{
  readonly attribute long max_size;
  readonly attribute unsigned long count;

  boolean enqueue(in WorkItem work, in long timeout);

  boolean enqueue_immediate(in WorkItem work);

  boolean is_full();

  boolean is_empty();

  boolean activate();

  boolean deactivate();

  boolean owns_current_thread();

  void flush();
};
```

WorkQueue types

You can implement your own `WorkQueue` interface, or use IONA-supplied `WorkQueue` factories to create one of two `WorkQueue` types:

- ManualWorkQueue
- AutomaticWorkQueue

ManualWorkQueue

Overview

A `ManualWorkQueue` is a work queue that holds incoming requests until they are explicitly dequeued. It allows the developer full control over how requests are processed by the POA.

IDL

The interface is defined as follows:

```
\\ IDL
interface ManualWorkQueue : WorkQueue
{
  boolean dequeue(out WorkItem work, in long timeout);

  boolean do_work(in long number_of_jobs, in long timeout);

  void shutdown(in boolean process_remaining_jobs);
};
```

Creating

You create a `ManualWorkQueueFactory` by calling `resolve_initial_references("IT_ManualWorkQueueFactory")`. The `ManualWorkQueueFactory` has the following interface:

```
interface ManualWorkQueueFactory
{
  ManualWorkQueue create_work_queue(in long max_size);
};
```

`create_work_queue` takes the following argument:

max_size is the maximum number of work items that the queue can hold. If the queue becomes full, the transport considers the server to be overloaded and tries to gracefully close down connections to reduce the load.

How requests are processed

Applications that use a `ManualWorkQueue` must periodically call `dequeue()` or `do_work()` to ensure that requests are processed. The developer is in full control of time between calls and if the events are processed by multiple

threads or in a single thread. If the developer chooses a multithreaded processing method, they are responsible for ensuring that the code is thread safe.

A false return value from either `do_work()` or `dequeue()` indicates that the timeout for the request has expired or that the queue has shut down.

AutomaticWorkQueue

Overview

An `AutomaticWorkQueue` is a work queue that feeds a thread pool. Automatic work queues process requests in the same way that the standard ORB does; however, it does allow the developer to assign a customized thread pool to a particular POA. Also, the developer can implement several automatic work queues to process different types of requests at different priorities.

IDL

The interface is defined as follows:

```
// IDL
interface AutomaticWorkQueue : WorkQueue
{
  readonly attribute unsigned long threads_total;
  readonly attribute unsigned long threads_working;

  attribute long high_water_mark;
  attribute long low_water_mark;

  void shutdown(in boolean process_remaining_jobs);
};
```

Creating

You create an `AutomaticWorkQueue` through the `AutomaticWorkQueueFactory`, obtained by calling `resolve_initial_references("IT_AutomaticWorkQueue")`. The `AutomaticWorkQueueFactory` has the following interface:

```
interface AutomaticWorkQueueFactory
{
  AutomaticWorkQueue create_work_queue(
        in long          max_size,
        in unsigned long initial_thread_count,
        in long          high_water_mark,
        in long          low_water_mark);

  AutomaticWorkQueue create_work_queue_with_thread_stack_size(
        in long          max_size,
        in unsigned long initial_thread_count,
        in long          high_water_mark,
        in long          low_water_mark,
        in long          thread_stack_size);
};
```

`create_work_queue()` takes these arguments:

max_size is the maximum number of work items that the queue can hold. To specify an unlimited queue size, supply a value of `-1`.

initial_thread_count is the initial number of threads in the thread pool; the ORB automatically creates and starts these threads when the workqueue is created.

high_water_mark specifies the maximum number of threads that can be created to process work queue items. To specify an unlimited number of threads, supply a value of `-1`.

low_water_mark lets the ORB remove idle threads from the thread pool, down to the value of `low_water_mark`. The number of available threads is never less than this value.

If you wish to have greater control of the size of the work queue's thread stack, use `create_work_queue_with_thread_stack()`. It adds one argument, `thread_stack_size`, to the end of the argument list. This argument specifies the size of the workqueues thread stack.

How requests are processed

Applications that use an `AutomaticWorkQueue` do not need to explicitly dequeue work items; instead, work items are automatically dequeued and processed by threads in the thread pool.

If all threads are busy and the number of threads is less than `high_water_mark`, the ORB can start additional threads to process items in the work queue, up to the value of `high_water_mark`. If the number of threads is equal to `high_water_mark` and all are busy, and the work queue is filled to capacity, the transport considers the server to be overloaded and tries to gracefully close down connections to reduce the load.

Using a WorkQueue

Creating the WorkQueue

To create a POA with a `WorkQueue` policy, follow these steps:

1. Create a work queue factory by calling `resolve_initial_references()` with the desired factory type by supplying an argument of `IT_AutomaticWorkQueueFactory` or `IT_ManualWorkQueueFactory`.
2. Set work queue parameters.
3. Create the work queue by calling `create_work_queue()` on the work queue factory.
4. Insert the work queue into an `Any`.
5. Add a work queue policy object to a POA's `PolicyList`.

Example 13 illustrates these steps:

Example 13: *Creating a POA with a WorkQueue policy*

```
import com.iona.corba.IT_WorkQueue.*;
import com.iona.corba.IT_PortableServer.*;

1  // get an automatic work queue factory
org.omg.CORBA.Object obj = orb.resolve_initial_references(
    "IT_AutomaticWorkQueueFactory");
AutomaticWorkQueueFactory wqf =
    AutomaticWorkQueueFactoryHelper.narrow(obj);

2  // set work queue parameters
int max_size = 20;
int init_thread_count = 1;
int high_water_mark = 10;
int low_water_mark = 2;

3  // create work queue
AutomaticWorkQueue auto_wq = wqf.create_work_queue(
    max_size,
    init_thread_count,
    high_water_mark,
    low_water_mark);
```

Example 13: *Creating a POA with a WorkQueue policy*

```
4  // insert the work queue into an any
   org.omg.CORBA.Any work_queue_policy_val = orb.create_any();
   work_queue_policy_val.insert_Object(auto_wq);

   // set other POA policies set
   // ...

   // create PolicyList
   org.omg.CORBA.Policy[] policies=new org.omg.CORBA.Policy[];

5  // add work queue policy object to POA's PolicyList
   orb.create_policy(DISPATCH_WORKQUEUE_POLICY_ID.value,
      work_queue_policy_val),

   // add other POA policies to PolicyList
   // ...
```

Processing events in a manual work queue

When using a manual work queue, the developer must implement the loop which removes requests from the queue.

Example 14 demonstrates one way to remove requests from a manual work queue. The code loops indefinitely and continuously polls the queue for requests. When there are requests on the queue, they are removed from the queue using the `dequeue()` method and then they processed with the `execute()` method of the `WorkItem` object returned from `dequeue()`.

Example 14: *Removing requests from a work queue.*

```
WorkQueue::WorkItem work_item;

while (1)
{
  if (wq.is_empty())
    {
     // Since there are no requests to process
     // the object can sleep, or do whatever other work
     // the developer needs done.
     ....
    }
  else
    {
      manual_work_queue.dequeue(work_item, 5000);
      work_item.execute();
      // no need to explicitly destroy as execute deletes the
      // work item once completed.
    }
}
```

Alternatively, you remove requests from the queue using the `do_work()` method. The difference is that using `do_work()` you can process several requests at one time.

Processing events in an automatic work queue

Automatic work queues handle request processing under the covers. Therefore, the developer does not need to implement any request handling logic.

Controlling POA Proxification

Overview

The Iona firewall proxy service, if it is activated, default behavior is to proxify all POAs. This can consume resources and degrade performance of a system if a large number of POAs are placed behind the firewall proxy service. In many instances only specific POAs will need to face outside the firewall. Using the `InterdictionPoilcy` you can control if a specific POA is proxified.

Policy

The `InterdictionPolicy` controls the behavior of the firewall proxy service plug-in, if it is loaded. The policy has two settings:

`ENABLE`	This is the default behavior of the firewall proxy service plug-in. A POA with its `InterdictionPolicy` set to `ENABLE` will be proxified.
`DISABLE`	This setting tells the firewall proxy service plug-in to not proxify the POA. A POA with its `InterdictionPolicy` set to `DISABLE` will not use the firewall proxy service and requests made on objects under its control will come directly from the requesting clients.

Example

The following code samples demonstrate how to set the `InterdictionPolicy` on a POA. In the examples, the policy is set to `DISABLE`.

Java

```
import com.iona.corba.IT_FPS.*;

// Create a PREVENT interdiction policy.
Any interdiction = m_orb.create_any();
InterdictionPolicyValueHelper.insert(interdiction,
   InterdictionPolicyValue.DISABLE);

Policy[] policies = new Policy[1];
polices[0] = m_orb.create_policy(INTERDICTION_POLICY_ID.value,
   interdiction);

// Create and return new POA.
return m_poa.create_POA("no_fps_poa", null, policies);
```

CHAPTER 10

Developing a Client

A CORBA client initializes the ORB runtime, handles object references, invokes operations on objects, and handles exceptions that these operations throw.

In this chapter

This chapter covers the following topics:

For information about exception handling, see Chapter 12.

Mapping IDL Interfaces to Proxies

When you compile IDL, the compiler maps each IDL interface to a client-side Java interface of the same name. A class of the name *_interface-name*Stub implements this interface and acts as the client-side proxy for the corresponding server object. Proxy classes implement the client-side call stubs that marshal parameter values and send operation invocations to the correct destination object. When a client invokes on a proxy method that corresponds to an IDL operation, Orbix conveys the call to the corresponding server object, whether remote or local.

The client application accesses proxy methods only through an object reference. When the client brings an object reference into its address space, the client runtime ORB instantiates a proxy to represent the object. In other words, a proxy acts as a local ambassador for the remote object.

For example, interface `Bank::Acount` has this IDL definition:

```
module BankDemo
{
    typedef float CashAmount;
    exception InsufficientFunds {};
    // ...
    interface Account{
        void withdraw( in CashAmount amount )
        raises (InsufficientFunds);

        // ... other operations not shown
    };
};
```

Given this IDL, the IDL compiler generates the following definitions for the client implementation:

```
package BankDemo;

public interface AccountOperations {
    java.lang.String account_id();

    // ...
    void withdraw(float amount)
        throws BankDemo.AccountPackage.InsufficientFunds;

    // other operations not shown ...
}

package BankDemo;

public interface Account
    extends AccountOperations,
    org.omg.CORBA.Object,
    org.omg.CORBA.portable.IDLEntity {}

package BankDemo;

public class _AccountStub
    extends org.omg.CORBA.portable.ObjectImpl
    implements Account {
    public void withdraw(float amount)
        throws BankDemo.AccountPackage.InsufficientFunds {
    // implementation details not shown...
    }
}
```

This proxy class demonstrates several characteristics that are true of all proxy classes:

- Member methods derive their names from the corresponding interface operations—in this case, `withdrawal()`.
- The proxy class inherits from `org.omg.CORBA.portable.ObjectImpl`, so the client can access all the inherited functionality of a CORBA object.

Using Object References

For each IDL interface definition, a POA server can generate and export references to the corresponding object that it implements. To access this object and invoke on its methods, a client must obtain an object reference—generally, from a CORBA naming service.

Object Reference Operations

Because all object references inherit from `CORBA::Object`, you can invoke its operations on any object reference. `CORBA::Object` is a pseudo-interface with this definition:

```
module CORBA{    //PIDL
// ..
    interface Object{
        Object      duplicate()
        void        release();
        boolean     is_nil();
        boolean     is_a(in string repository_id);
        boolean     non_existent();
        boolean     is_equivalent(in Object other_object);
        boolean     hash(in unsigned long max);
        // ...
    }
};
```

Mappings

In Java, these operations are mapped to `org.omg.CORBA.Object` member methods as follows:

```
package org.omg.CORBA;
public interface Object {
    boolean _is_a(String Identifier);
    boolean _is_equivalent(Object that);
    boolean _non_existent();
    int _hash(int maximum);
    org.omg.CORBA.Object _duplicate();
    void _release();
   // ...
}
```

`_duplicate()` and `_release()` are not implemented in the Java version of Orbix. To duplicate an object reference In Java, simply assign the original object reference to the new reference. For example:

```
Account acct1 = ...;     // Get ref from somewhere...
Account acct2;           // acc2 has undefined contents
acc2 = acct1;            // Both reference same Account
```

Given JVM garbage collection, `_release()` is redundant.

Operation descriptions

The following sections describe the remaining operations.

_is_a() is similar to `narrow()` in that it lets you to determine whether an object supports a specific interface. For example:

```
org.omg.CORBA.Object obj = ...;    // Get a reference

if (obj != null && obj._is_a("IDL:BankDemo/Account:1.0"))
    // It's an Account object...
else
    // Some other type of object...
```

The test for `null` in this code example prevents the client program from making a call via a null object reference.

`_is_a()` lets applications manipulate IDL interfaces without static knowledge of the IDL—that is, without having linked the IDL-generated stubs. Most applications have static knowledge of IDL definitions, so they never need to call `_is_a()`. In this case, you can rely on the helper class's `narrow()` method to ascertain whether an object supports the desired interface.

_non_existent() tests whether a CORBA object exists. `_non_existent()` returns true if an object no longer exists. A return of true denotes that this reference and all copies are no longer viable and should be released.

If `_non_existent()` needs to contact a remote server, the operation is liable to raise system exceptions that have no bearing on the object's existence—for example, the client might be unable to connect to the server.

If you invoke a user-defined operation on a reference to a non-existent object, the ORB raises the `OBJECT_NOT_EXIST` system exception. So, invoking an operation on a reference to a non-existent object is safe, but the client must be prepared to handle errors.

_is_equivalent() tests whether two references are identical. If `_is_equivalent()` returns true, you can be sure that both references point to the same object.

A false return does not necessarily indicate that the references denote different objects, only that the internals of the two references differ in some way. The information in references can vary among different ORB implementations. For example, one vendor might enhance performance by adding cached information to references, to speed up connection establishment. Because `_is_equivalent()` tests for absolute identity, it cannot distinguish between vendor-specific and generic information.

_hash() returns a hash value in the range `0..max-1`. The hash value remains constant for the lifetime of the reference. Because the CORBA specifications offer no hashing algorithm, the same reference on different ORBs can have different hash values.

`_hash()` is guaranteed to be implemented as a local operation—that is, it will not send a message on the wire.

`_hash()` is mainly useful for services such as the transaction service, which must be able to determine efficiently whether a given reference is already a member of a set of references. `_hash()` permits partitioning of a set of references into an arbitrary number of equivalence classes, so set membership testing can be performed in (amortized) constant time. Applications rarely need to call this method.

Narrowing Object References

Before a client can use an object reference, it must narrow it to the appropriate interface. For each IDL interface, the IDL compiler generates a helper class with a `narrow()` method that returns the narrowed object reference. Each helper class name has the format *`Interface-name`*`Helper`, with the first letter capitalized.

For example, the IDL compiler generates the `AccountHelper` class for the `Account` interface. The `AccountHelper` class defines the following `narrow()` method to return `Account` objects:

```
public static BankDemo.Account narrow(
   org.omg.CORBA.Object _obj
) throws org.omg.CORBA.BAD_PARAM {
    if (_obj == null) {
        return null;
    }
    if (_obj instanceof BankDemo.Account) {
        return (BankDemo.Account)_obj;
    }
    if (_obj instanceof BankDemo._AccountStub) {
        return (BankDemo._AccountStub)_obj;
    }
    if (_obj._is_a(id())) {
        BankDemo._AccountStub _ref =
            new BankDemo._AccountStub();
        _ref._set_delegate(

   ((org.omg.CORBA.portable.ObjectImpl)_obj)._get_delegate());
        return _ref;
    }
    else {
        throw new org.omg.CORBA.BAD_PARAM();
    }
}
```

A client program might call this `narrow()` method as follows:

```
org.omg.CORBA.Object obj = ...; // get object reference somehow

// Narrow obj to an Account object
Account acct = AccountHelper.narrow(obj);

// do stuff with Account object
// ...
```

If the parameter that is passed to `AccountHelper.narrow()` is not of class `Account` or one of its derived classes, `Account.narrow()` raises the `CORBA.BAD_PARAM` exception.

String Conversions

Object references can be converted to and from strings, which facilitates persistent storage. When a client obtains a stringified reference, it can convert the string back into an active reference and contact the referenced object. The reference remains valid as long as the object remains viable. When the object is destroyed, the reference becomes permanently invalid.

Operations

The `object_to_string()` and `string_to_object()` operations are defined in Java as follows:

```
package org.omg.CORBA;

public abstract class ORB {

    public abstract org.omg.CORBA.Object
        string_to_object(String str);
    public abstract String
        object_to_string(org.omg.CORBA.Object obj);
```

object_to_string()

For example, the following code stringifies an `Account` object reference:

```
BankDemo.Account acct = ...;     // Account reference

// Write reference as a string to stdout
try {
    java.lang.String str = orb.object_to_string(acct);
    System.out.println(str);
} catch (Exception ex) {
    // Deal with error...
}
```

This code prints an IOR (interoperable reference) string whose format is similar to this:

```
IOR:
010000002000000049444c3a61636d652e636f6d2f4943532f436f6e74726f6c
c65723a312e3000010000000000000004a000000010102000e0000003139322e3
36382e312e3231300049051b0000003a3e0231310c01000000c7010000234800
008000000000000000000010000000600000006000000010000001100
```

The stringified references returned by `object_to_string()` always contain the prefix `IOR:`, followed by an even number of hexadecimal digits. Stringified references do not contain any unusual characters, such as control characters or embedded newlines, so they are suitable for text I/O.

---

string_to_object()

To convert a string back into a reference, call `string_to_object()`:

```
// Assume stringified reference is in accv[0]
try {
    org.omg.CORBA.Object obj;
    obj = orb.string_to_object(accv[0]);
    if (obj == null)
        System.exit(1);

    BankDemo.Account acct = BankDemo.AccountHelper.narrow(obj);
    if (acct = null)
        System.exit(1);    // Not an Account reference

    // Use acct reference
    // ...

} catch (Exception ex) {
    // Deal with error...
}
```

The CORBA specification defines the representation of stringified IOR references, so it is interoperable across all ORBs that support the Internet Inter-ORB Protocol (*IIOP*).

Although the IOR shown earlier looks large, its string representation is misleading. The in-memory representation of references is much more compact. Typically, the incremental memory overhead for each reference in a client can be as little as 30 bytes.

You can also stringify or destringify a null reference. Null references look like one of the following strings:

```
IOR:00000000000000010000000000000000
IOR:01000000010000000000000000000000
```

Constraints

IOR string references should be used only for these tasks:

- Store and retrieve an IOR string to and from a storage medium such as disk or tape.
- Conversion to an active reference.

It is inadvisable to rely on IOR string references as database keys for the following reasons:

- Actual implementations of IOR strings can vary across different ORBs—for example, vendors can add proprietary information to the string, such as a time stamp. Given these differences, you cannot rely on consistent string representations of any object reference.
- The actual size of IOR strings—between 200 and 600 bytes— makes them prohibitively expensive to use as database keys.

In general, you should not compare one IOR string to another. To compare object references, use `is_equivalent()` (see page 238).

Note: Stringified IOR references are one way to make references to initial objects known to clients. However, distributing strings as e-mail messages or writing them into shared file systems is neither a distributed nor a scalable solution. More typically, applications obtain object references through the naming service (see Chapter 17 on page 421).

Using corbaloc URL strings

`string_to_object()` can also take as an argument a corbaloc-formatted URL, and convert it into an object reference. A corbaloc URL denotes objects that can be contacted by IIOP or `resolve_initial_references()`.

A corbaloc URL uses one of the following formats:

```
corbaloc:rir:/rir-argument
corbaloc:iiop-address[, iiop-address].../key-string
```

rir-argument: A value that is valid for `resolve_initial_references()`, such as `NameService`.

iiop-address: Identifies a single IIOP address with the following format:

[`iiop`*]:[major-version-num.minor-version-num@]host-spec[*`:`*port-num]*

IIOP version information is optional; if omitted, version 1.0 is assumed. *`host-spec`* can specify either a DNS-style host name or a numeric IP address; specification of *`port-num`* is optional.

key-string: corresponds to the octet sequence in the object key member of a stringified object reference, or an object's named key that is defined in the implementation repository.

For example, if you register the named key `BankService` for an IOR in the implementation repository, a client can access an object reference with `string_to_object()` as follows:

```
// assume that xyz.com specifies a location domain's host
global_orb.string_to_object
    ("corbaloc:iiop:xyz.com/BankService");
```

The following code obtains an object reference to the naming service:

```
global_orb.string_to_object("corbaloc:rir:/NameService");
```

You can define a named key in the implementation repository through the `•itadmin named_key create` command. For more information, see the *Application Server Platform Administrator's Guide*.

Initializing and Shutting Down the ORB

Before a client application can start any CORBA-related activity, it must initialize the ORB runtime by calling `org.omg.CORBA.ORB.init()`. `ORB.init()` returns an object reference to the ORB object; this, in turn, lets the client obtain references to other CORBA objects, and make other CORBA-related calls.

Procedures for ORB initialization and shutdown are the same for both servers and clients. For detailed information, see "ORB Initialization and Shutdown" on page 155.

Invoking Operations and Attributes

For each IDL operation in an interface, the IDL compiler generates a method with the name of the operation in the corresponding proxy. It also maps each unqualified attribute to a pair of overloaded methods with the name of the attribute, where one method acts as an accessor and the other acts as a modifier. For `readonly` attributes, the compiler generates only an accessor method.

An IDL attribute definition is functionally equivalent to a pair of set/get operation definitions, with this difference: attribute accessors and modifiers can only raise system exceptions, while user exceptions apply only to operations.

For example, the following IDL defines a single attribute and two operations in interface `Test::Example`:

```
module Test {

    interface Example {
        attribute string name;
        oneway void set_address(in string addr);
        string get_address();
    };
};
```

The IDL compiler maps this definition's members to the following methods in the Java interface `ExampleOperations`, which is inherited by the generated interface `Example`. A client invokes on these methods as if their implementations existed within its own address space:

```
package Test;

public interface ExampleOperations {
     java.lang.String name();
     void name(java.lang.String _val);
     void set_address(java.lang.String addr);
     java.lang.String get_address();
}

package Test;

public interface Example
    extends ExampleOperations,
    org.omg.CORBA.Object,
    org.omg.CORBA.portable.IDLEntity {}
```

Passing Parameters in Client Invocations

IDL `in` parameters always map directly to the corresponding Java type. This mapping is possible because `in` parameters are always passed by value, and Java supports by-value passing of all types. Similarly, IDL return values always map directly to the corresponding Java type.

IDL `inout` and `out` parameters must be passed by reference because they might be modified during an operation call, and do not map directly into the Java parameter passing mechanism. In the IDL-to-Java mapping, IDL `inout` and `out` parameters are mapped to Java holder classes. Holder classes simulate passing by reference. For each IDL `out` or `inout` parameter, the client supplies an instance of the appropriate Java holder class. The contents of the holder instance are modified by the call, and the client uses the contents when the call returns.

Holder Class Types

There are two categories of holder classes:

- Holders for basic types.
- Holders for user-defined types.

Holders for basic types

Holder classes for basic Java types and the Java `string` type are available in the package `org.omg.CORBA`. Each holder class is named *Type*`Holder`, where *Type* is the name of a basic Java type with the first letter capitalized. For example, the `IntHolder` class is defined for `int` types as follows:

```
package org.omg.CORBA;

import org.omg.CORBA.portable.Streamable;
import org.omg.CORBA.portable.InputStream;
import org.omg.CORBA.portable.OutputStream;

public final class IntHolder implements Streamable {

    public int value;

    public IntHolder() {
    }
    public IntHolder(int initial) {
        value = initial;
    }
    public void _read(InputStream input) {
        value = input.read_long();
    }
    public void _write(OutputStream output) {
        output.write_long(value);
    }
    public org.omg.CORBA.TypeCode _type() {
        return ORB.init().get_primitive_tc(TCKind.tk_long);
    }
}
```

Holders for user-defined types

Holder classes for user-defined types, including IDL interface types, are generated by the Java mapping. The name format is *type*`Holder`. For example, given the user-defined interface `Account`, the following holder class is generated:

```
public final class AccountHolder
    implements org.omg.CORBA.portable.Streamable {

    public BankDemo.Account value;

    public AccountHolder() {}
    public AccountHolder(BankDemo.Account value) {
        this.value = value;
    }
    public void _read(
        org.omg.CORBA.portable.InputStream _stream) {
        value = BankDemo.AccountHelper.read(_stream);
    }
    public void _write(
        org.omg.CORBA.portable.OutputStream _stream) {
        BankDemo.AccountHelper.write(_stream, value);
    }
    public org.omg.CORBA.TypeCode _type() {
        return BankDemo.AccountHelper.type();
    }
}
```

Holder Class Members

All Holder classes, basic and user-defined, contain these members:

- `value` contains the value that the client supplies as the `out` or `inout` argument, and which the server runtime ORB resets before the operation returns.
- A constructor for `out` parameters that initializes a new holder object's `value` to empty—for example, 0 for numeric types and null for string.
- A constructor for `inout` parameters that initializes a new holder object's value field to the supplied argument's value.
- `_read()` reads unmarshalled data from input and assigns it to `value`.
- `_write()` marshals the data in `value`.
- `_type()` returns the `TypeCode` object that corresponds to `value's` data type.

Invoking an Operation With Holder Classes

A client that invokes operations with `inout` and `out` parameters must supply holder objects as arguments for those parameters. For each IDL `out` or `inout` parameter, the client program instantiates the appropriate holder class and supplies it to the method call. On return, the client can evaluate the contents of the holder object.

- For `out` parameters, the client calls the default constructor, so the holder object's value field is initialized to empty. The servant's implementation of the invoked operation resets the holder object's value on return.
- For `inout` parameters, the client initializes each holder with a valid value by calling the appropriate constructor.

For example, the following IDL modifies the `create_account()` operation in the `BankDemo` module so that it supplies two `out` parameters, the account object and the new account's ID:

```
// IDL
module BankDemo {
    // ...

    // Forward declaration of Account
    interface Account;

    interface Bank {
        void create_account(
            (in string name, out Account acct, out string acc_id)
        // ...
    }
    // ...
}
```

The IDL compiler maps `create_account()` to the Java method `BankDemo.Bank.create_account()`; the operation's two `out` parameters require two holder classes: `AccountHolder` and `StringHolder`, so the client program can pass these two parameters by reference to a `Bank` object servant:

```
public void create_account(
    java.lang.String name, BankDemo.AccountHolder acct,
    org.omg.CORBA.StringHolder acc_id);
```

The servant sets the two `out` parameter values; when the invocation returns, the client receives these values.

The following client code shows how this operation might be invoked:

```
import org.omg.CORBA.SystemException;

public class client {

org.omg.CORBA.ORB global_orb;

public static void main (java.lang.String[] args) {
    try {
        global_orb = org.omg.CORBA.ORB.init(args, null);
        Bank b_ref = ...; // get object reference to bank
        if (b_ref == null)
            System.exit(1);
    }

        // create holder objects
        StringHolder acct_id_holder = new StringHolder();
        AccountHolder acct_holder = new AccountHolder();

        try {
            // create a bank account
            b_ref.create_account(
                "Joe", acct_holder, acct_id_holder);
        }
        catch (SystemException se) {
            System.out.println ("Unexpected exception on bind");
           System.out.println ("Exception: " + se);
           System.exit(1);
        }

        // Retrieve values from holder objects
        Account acct_ref = acct_holder.value;
        java.lang.String acct_id = acct_id_holder.value

        try {
            // invoke operations on account object
            // not shown ...
        }
        catch (SystemException se) {
            // catch clauses not shown ...
        }
}
```

In the server, the servant implementation of `create_account()` receives the holder object for type `Account` and can manipulate its `value` field. For information about how servants handle `out` and `in/out` parameter types, see "Handling Output Parameters" on page 187.

Client Policies

Orbix supports a number of quality of service policies, which can give a client programmatic control over request processing:

- RebindPolicy specifies whether the ORB transparently reopens closed connections and rebinds forwarded objects.
- SyncScopePolicy determines how quickly a client resumes processing after sending one-way requests.
- Timeout policies offer different degrees of control over the length of time that an outstanding request remains viable.

You can set quality of service policies at three scopes, in descending order of precedence:

1. On individual objects, so they apply only to invocations on those objects.
2. On a given thread, so they apply only to invocations on that thread
3. On the client ORB, so they apply to all invocations.

You can set policies in any combination at all three scopes; the *effective* policy is determined on each invocation. If settings are found for the same policy type at more than one scope, the policy at the lowest scope prevails.

For detailed information about setting these and other policies on a client, see "Setting Client Policies" on page 170.

RebindPolicy

A client's `RebindPolicy` determines whether the ORB can transparently reconnect and rebind. A client's rebind policy is set by a `RebindMode` constant, which describes the level of transparent binding that can occur when the ORB tries to carry out a remote request:

TRANSPARENT The default policy: the ORB silently reopens closed connections and rebinds forwarded objects.

NO_REBIND The ORB silently reopens closed connections; it disallows rebinding of forwarded objects if client-visible policies have changed since the original binding. Objects can be explicitly rebound by calling `CORBA::Object::validate_connection()` on them.

NO_RECONNECT The ORB disallows reopening of closed connections and rebinding of forwarded objects. Objects can be explicitly rebound by calling `CORBA::Object::validate_connection()` on them.

Note: Currently, Orbix requires rebinding on reconnection. Therefore, `NO_REBIND` and `NO_RECONNECT` policies have the same effect.

SyncScopePolicy

A client's `SyncScopePolicy` determines how quickly it resumes processing after sending one-way requests. You specify this behavior with one of these `SyncScope` constants:

SYNC_NONE The default policy: Orbix clients resume processing immediately after sending one-way requests, without knowing whether the request was processed, or whether it was even sent over the wire.

SYNC_WITH_TRANSPORT The client resumes processing after a transport accepts the request. This policy is especially helpful when used with store-and-forward transports. In that case, this policy offer clients assurance of a high degree of probable delivery.

SYNC_WITH_SERVER The client resumes processing after the request finds a server object to process it—that is, the server ORB sends a `NO_EXCEPTION` reply. If the request must be forwarded, the client continues to block until location forwarding is complete.

SYNC_WITH_TARGET The client resumes processing after the request processing is complete. This behavior is equivalent to a synchronous (two-way) operation. With this policy in effect, a client has absolute assurance that a its request has found a target and been acted on. The object transaction service (OTS) requires this policy for any operation that participates in a transaction.

Note: This policy only applies to GIOP 1.2 (and higher) requests.

Timeout Policies

A responsive client must be able to specify timeouts in order to abort invocations. Orbix supports several standard OMG timeout policies, as specified in the Messaging module; it also provides proprietary policies in the `IT_CORBA` module that offer more fine-grained control. Table 11 shows which policies are supported in each category:

Table 11: *Timeout Policies*

OMG Timeout Policies	`RelativeRoundtripTimeoutPolicy`
	`ReplyEndTimePolicy`
	`RelativeRequestTimeoutPolicy`
	`RequestEndTimePolicy`

Proprietary Timeout Policies	`BindingEstablishmentPolicy`
	`RelativeBindingExclusiveRoundtripTimeoutPolicy`
	`RelativeBindingExclusiveRequestTimeoutPolicy`
	`RelativeConnectionCreationTimeoutPolicy`
	`InvocationRetryPolicy`

If a request's timeout expires before the request can complete, the client receives the system exception `CORBA::TIMEOUT`.

Note: When using these policies, be careful that their settings are consistent with each other. For example, the `RelativeRoundtripTimeoutPolicy` specifies the maximum amount of time allowed for round-trip execution of a request.

Orbix also provides its own policies, which let you control specific segments of request execution—for example, `BindingEstablishmentPolicy` lets you set the maximum time to establish bindings.

It is possible to set the maximum binding time to be greater than the maximum allowed for roundtrip request execution. Although these settings are inconsistent, no warning is issued; and Orbix silently adheres to the more restrictive policy.

Setting absolute and relative times

Two policies, `RequestEndTimePolicy` and `ReplyEndTimePolicy`, set absolute deadlines for request and reply delivery, respectively, through the `TimeBase::UtcT` type. Other policies set times that are relative to a specified

event—for example, `RelativeRoundtripTimeoutPolicy` limits how much time is allowed to deliver a request and its reply, starting from the request invocation.

Orbix libraries include the `com.iona.common.time.UTCUtility` helper class, which provides static utility methods for working with the types defined in the `TimeBase` module. For example, `future_time()` lets you get an absolute time that is relative to the current time.

You can specify absolute times in long epoch (15 Oct. 1582 to ~30000AD) Universal Time Coordinated (UTC), or relative times in 100 nano-seconds units using the OMG Time Service's `TimeBase::UtcT` type. You can also convert times to short epoch (Jan. 1 1970 to ~2038) UTC in millisecond units. All times created have zero displacement from GMT.

For more information, refer to the *CORBA Programmer's Reference*.

Imported Java packages

Programs that use timeout policies typically include the following `import` statements:

```
import org.omg.Messaging.*;
import org.omg.Timebase.*;
import com.iona.corba.IT_CORBA.*;
```

The examples that follow all assume that these packages are imported.

Policies

RelativeRoundtripTimeoutPolicy specifies how much time is allowed to deliver a request and its reply. Set this policy's value in 100-nanosecond units. No default is set for this policy; if it is not set, a request has unlimited time to complete.

The timeout countdown begins with the request invocation, and includes the following activities:

- Marshalling in/inout parameters
- Any delay in transparently establishing a binding

If the request times out before the client receives the last fragment of reply data, the request is cancelled via a GIOP `CancelRequest` message and all received reply data is discarded.

For example, the following code sets a `RelativeRoundtripTimeoutPolicy` override on the ORB PolicyManager, setting a four-second limit on the time allowed to deliver a request and receive the reply:

```
long relative_expiry = 4L * 10000000L; // 4 seconds
try{
   Any relative_roundtrip_timeout_value = orb.create_any();
   TimeTHelper.insert(
      relative_roundtrip_timeout_value,
      relative_expiry
   );
   Policy [] policies = new Policy[1];
   policies[0] = orb.create_policy(
                        RELATIVE_RT_TIMEOUT_POLICY_TYPE.value,
                        relative_roundtrip_timeout_value);
   policy_manager.set_policy_overrides(
                        policies,
                        SetOverrideType.ADD_OVERRIDE);
}
catch(PolicyError pe){
   System.exit(1);
}
catch(InvalidPolicies ip){
   System.exit(1);
}
catch(SystemException se){
   System.exit(1);
}
```

ReplyEndTimePolicy sets an absolute deadline for receipt of a reply. This policy is otherwise identical to `RelativeRoundtripTimeoutPolicy`. Set this policy's value with a `TimeBase::UtcT` type (see "Setting absolute and relative times" on page 259).

No default is set for this policy; if it is not set, a request has unlimited time to complete.

RelativeRequestTimeoutPolicy specifies how much time is allowed to deliver a request. Request delivery is considered complete when the last fragment of the GIOP request is sent over the wire to the target object. The timeout-specified period includes any delay in establishing a binding. This policy type is useful to a client that only needs to limit request delivery time. Set this policy's value in 100-nanosecond units.

No default is set for this policy; if it is not set, request delivery has unlimited time to complete.

For example, the following code sets a `RelativeRequestTimeoutPolicy` override on the ORB PolicyManager, setting a three-second limit on the time allowed to deliver a request:

```
long relative_expiry = 3L * 10000000L; // 3 seconds
try{
   Any relative_request_timeout_value = orb.create_any();

   TimeTHelper.insert(
      relative_request_timeout_value,
      relative_expiry);
   Policy [] policies = new Policy[1];
   policies[0] = orb.create_policy(
                        RELATIVE_REQ_TIMEOUT_POLICY_TYPE.value,
                        relative_request_timeout_value);
   policy_manager.set_policy_overrides(
                        policies,
                        SetOverrideType.ADD_OVERRIDE);
}
catch(PolicyError pe){
   System.exit(1);
}
catch(InvalidPolicies ip){
   System.exit(1);
}
catch(SystemException se){
   System.exit(1);
}
```

RequestEndTimePolicy sets an absolute deadline for request delivery. This policy is otherwise identical to `RelativeRequestTimeoutPolicy`. Set this policy's value with a `TimeBase::UtcT` type (see "Setting absolute and relative times" on page 259).

No default is set for this policy; if it is not set, request delivery has unlimited time to complete.

BindingEstablishmentPolicy limits the amount of effort Orbix puts into establishing a binding. The policy equally affects transparent binding (which results from invoking on an unbound object reference), and explicit binding (which results from calling `Object::_validate_connection()`.

A client's `BindingEstablishmentPolicy` is determined by the members of its `BindingEstablishmentPolicyValue`, which is defined as follows:

```
struct BindingEstablishmentPolicyValue
{
    TimeBase::TimeT relative_expiry;
    unsigned short  max_binding_iterations;
    unsigned short  max_forwards;
    TimeBase::TimeT initial_iteration_delay;
    float           backoff_ratio;
};
```

- `relative_expiry` limits the amount of time allowed to establish a binding. Set this member in 100-nanosecond units. The default value is infinity.
- `max_binding_iterations` limits the number of times the client tries to establish a binding. Set to -1 to specify unlimited retries. The default value is 5.

  Note: If location forwarding requires that a new binding be established for a forwarded IOR, only one iteration is allowed to bind the new IOR. If the first binding attempt fails, the client reverts to the previous IOR. This allows a load balancing forwarding agent to redirect the client to another, more responsive server.

- `max_forwards` limits the number of forward tries that are allowed during binding establishment. Set to -1 to specify unlimited forward tries. The default value is 20.
- `initial_iteration_delay` sets the amount of time, in 100-nanosecond units, between the first and second tries to establish a binding. The default value is 0.1 seconds.

- `backoff_ratio` lets you specify the degree to which delays between binding retries increase from one retry to the next. The successive delays between retries form a geometric progression:

  ```
  0,
  initial_iteration_delay x backoff_ratio^0,
  initial_iteration_delay x backoff_ratio^1,
  initial_iteration_delay x backoff_ratio^2,
  ...,
  initial_iteration_delay x backoff_ratio^(max_binding_iterations -
     2)
  ```

  The default value is 2.

For example, the following code sets an `BindingEstablishmentPolicy` override on an object reference:

```
// ...
import com.iona.corba.util.ObjectHelper

try
{
   Any bind_est_value = orb.create_any();
   BindingEstablishmentPolicyValueHelper.insert(
      bind_est_value,
      new BindingEstablishmentPolicyValue(
                        (long) 30 * 10000000;  // 30 seconds
                        (short)5,         // 5 binding tries
                        (short)20,        // 20 forwards
                        (long)1000000,    // 0.1s delay
                        (float)2.0)     // back-off ratio
   );
   Policy [] policies = new Policy[1];
   policies[0] = orb.create_policy(
                        BINDING_ESTABLISHMENT_POLICY_ID.value,
                        bind_est_value);

   org.omg.CORBA.Object o =
   ObjectHelper.set_policy_overrides(obj_ref,
                                     policies,
                                   SetOverrideType.ADD_OVERRIDE);
}
```

```
catch(PolicyError pe){
   System.exit(1);
}
catch(InvalidPolicies ip){
   System.exit(1);
}
catch (SystemException se){
   System.exit(1);
}
```

RelativeBindingExclusiveRoundtripTimeoutPolicy limits the amount of time allowed to deliver a request and receive its reply, exclusive of binding attempts. The countdown begins immediately after a binding is obtained for the invocation. This policy's value is set in 100-nanosecond units.

RelativeBindingExclusiveRequestTimeoutPolicy limits the amount of time allowed to deliver a request, exclusive of binding attempts. Request delivery is considered complete when the last fragment of the GIOP request is sent over the wire to the target object. This policy's value is set in 100-nanosecond units.

RelativeConnectionCreationTimeoutPolicy specifies how much time is allowed to resolve each address in an IOR, within each binding iteration. Defaults to 8 seconds.

An IOR can have several `TAG_INTERNET_IOP` (IIOP transport) profiles, each with one or more addresses, while each address can resolve via DNS to multiple IP addresses. Furthermore, each IOR can specify multiple transports, each with its own set of profiles.

This policy applies to each IP address within an IOR. Each attempt to resolve an IP address is regarded as a separate attempt to create a connection. The policy's value is set in 100-nanosecond units.

InvocationRetryPolicy applies to invocations that receive the following exceptions:

- A `TRANSIENT` exception with a completion status of `COMPLETED_NO` triggers a transparent reinvocation.
- A `COMM_FAILURE` exception with a completion status of `COMPLETED_NO` triggers a transparent rebind attempt.

A client's `InvocationRetryPolicy` is determined by the members of its `InvocationRetryPolicyValue`, which is defined as follows:

```
struct InvocationRetryPolicyValue
{
    unsigned short  max_retries;
    unsigned short  max_rebinds;
    unsigned short  max_forwards;
    TimeBase::TimeT initial_retry_delay;
    float           backoff_ratio;
};
```

- max_retries limits the number of transparent reinvocation that are attempted on receipt of a `TRANSIENT` exception. The default value is 5.
- max_rebinds limits the number of transparent rebinds that are attempted on receipt of a `COMM_FAILURE` exception. The default value is 5.

  Note: This setting is valid only if the effective RebindPolicy is `TRANSPARENT`; otherwise, no rebinding occurs.

- max_forwards limits the number of forward tries that are allowed for a given invocation. Set to -1 to specify unlimited forward tries. The default value is 20.
- initial_retry_delay sets the amount of time, in 100-nanosecond units, between the first and second retries. The default value is 0.1 seconds.

  Note: The delay between the initial invocation and first retry is always 0.

  This setting only affects the delay between transparent invocation retries; it has no affect on rebind or forwarding attempts.

- backoff_ratio lets you specify the degree to which delays between invocation retries increase from one retry to the next. The successive delays between retries form a geometric progression:

  ```
  0,
  initial_iteration_delay x backoff_ratio^0,
  initial_iteration_delay x backoff_ratio^1,
  initial_iteration_delay x backoff_ratio^2,
  ...,
  initial_iteration_delay x backoff_ratio^(max_retries - 2)
  ```

  The default value is 2.

For example, the following code sets an `InvocationRetryPolicy` override on an object reference:

```
// ...
import com.iona.corba.util.ObjectHelper

try
{
   Any no_retries_value = orb.create_any();
   InvocationRetryPolicyValueHelper.insert(
      no_retries_value,
      new InvocationRetryPolicyValue(
                        (short)0,         // 0 retries
                        (short)5,         // 5 rebinds
                        (short)20,        // 20 forwards
                        (long)1000000,    // 0.1s delay
                        (float)2.0));     // back-off ratio
   Policy [] policies = new Policy[1];
   policies[0] = orb.create_policy(
                        INVOCATION_RETRY_POLICY_ID.value,
                        no_retries_value);
   org.omg.CORBA.Object o =
   ObjectHelper.set_policy_overrides(obj_ref,
                                     policies,
                                  SetOverrideType.ADD_OVERRIDE);
}
```

```
catch(PolicyError pe){
   System.exit(1);
}
catch(InvalidPolicies ip){
   System.exit(1);
}
catch (SystemException se){
   System.exit(1);
}
```

Implementing Callback Objects

Many CORBA applications implement callback objects on a client so that a server can notify the client of some event. You implement a callback object on a client exactly as you do on a server, by activating it in a client-side POA (see "Activating CORBA Objects" on page 185). This POA's LifeSpanPolicy should be set to `TRANSIENT`. Thus, all object references that the POA exports are valid only as long as the POA is running. This ensures that a late server callback is not misdirected to another client after the original client shuts down.

It is often appropriate to use a client's root POA for callback objects, inasmuch as it always exports transient object references. If you do so, make sure that your callback code is thread-safe; otherwise, you must create a POA with policies of `SINGLE_THREAD_MODEL` and `TRANSIENT`.

Managing Servants

A POA that needs to manage a large number of objects can be configured to incarnate servants only as they are needed. Alternatively, a POA can use a single servant to service all requests.

A POA's default request processing policy is `USE_ACTIVE_OBJECT_MAP_ONLY`. During POA initialization, the active object map must be populated with all object-servant mappings that are required during the POA's lifetime. The active object map maintains object-servant mappings until the POA shuts down, or an object is explicitly deactivated.

For example, you might implement the `BankDemo::Account` interface so that at startup, a server instantiates a servant for each account and activates all the account objects. Thus, a servant is always available for any client invocation on that account—for example, `balance()` or `withdraw()`.

Drawbacks of active object map usage

Given the potential for many thousands of accounts, and the likelihood that account information changes—accounts are closed down, new accounts are created—the drawbacks of this static approach become obvious:

- Code duplication: For each account, the same code for servant creation and activation must be repeated, increasing the potential for errors.
- Inflexibility: For each change in account information, you must modify and recompile the server code, then stop and restart server processes.

- Startup time: The time required to create and activate a large number of servants prolongs server startup and delays its readiness to process client requests.
- Memory usage: An excessive amount of memory might be required to maintain all servants continuously.

This scenario makes it clear that you should usually configure a POA to rely exclusively on an active object map only when it maintains a small number of objects.

Policies for managing many objects

If a POA is required to maintain a large number of objects, you should set its request processing policy to one of the following:

- `USE_SERVANT_MANAGER` specifies that servants are instantiated on demand.
- `USE_DEFAULT_SERVANT` specifies a default servant that handles requests for any objects that are not registered in the active object map, or for all requests in general.

This chapter shows how to implement both policies.

In this chapter

This chapter contains the following sections:

Using Servant Managers

Servant manager types

A POA whose request processing policy is set to `USE_SERVANT_MANAGER` supplies servants on demand for object requests. The POA depends on a servant manager to map objects to servants. Depending on its servant retention policy, the POA can implement one of two servant manager types, either a *servant activator* or *servant locator*:

- A servant activator is registered with a POA that has a `RETAIN` policy. The servant activator supplies a servant for an inactive object on receiving an initial request for it. The active object map retains the mapping between the object and its servant until the object is deactivated.
- A servant locator is registered with a POA that has a policy of `NON_RETAIN`. The servant locator supplies a servant for an inactive object each time the object is requested. In the absence of an active object map, the servant locator must deactivate the object and delete the servant from memory after the request returns.

Because a servant activator depends on the active object map to maintain the servants that it supplies, its usefulness is generally limited to minimizing an application's startup time. In almost all cases, you should use a servant locator for applications that must dynamically manage large numbers of objects.

Registering a servant manager

An application registers its servant manager —whether activator or locator— with the POA by calling `set_servant_manager()` on it; otherwise, an `OBJ_ADAPTER` exception is returned to the client on attempts to invoke on one of its objects.

The following sections show how to implement the `BankDemo::Account` interface with a servant activator and a servant locator. Both servant manager types activate account objects with instantiations of servant class `AccountImpl`, which inherits from skeleton class `AccountPOA`:

```
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import java.io.*;

import demos.servant_management.BankDemo.AccountPackage.*;
import demos.servant_management.BankDemo.*;

public class AccountImpl
extends AccountPOA
{
  public AccountImpl(String account_id,
                     AccountDatabase account_db)
  {
    m_account_db = account_db;
    m_account_id = account_id;
    m_balance = m_account_db.read_account(m_account_id);
  }

  protected void finalize()
  {
    m_account_db.write_account(m_account_id, m_balance);
  }

  protected void save_all()
  {
    m_account_db.write_account(m_account_id, m_balance);
  }

  public void withdraw(float amount) throws InsufficientFunds
  {
    if (amount > m_balance)
    {
      throw new InsufficientFunds();
    }
    m_balance -= amount;
  }

  public void deposit(float amount)
  {
    m_balance += amount;
  }

  public String account_id()
  {
    return m_account_id;
  }
```

```
  public float balance()
  {
    return m_balance;
  }

  private String m_account_id;
  private float m_balance;
  private AccountDatabase m_account_db;
}
```

Servant Activators

A POA with policies of `USE_SERVANT_MANAGER` and `RETAIN` uses a servant activator as its servant manager. The POA directs the first request for an inactive object to the servant activator. If the servant activator returns a servant, the POA associates it with the requested object in the active object map and thereby activates the object. Subsequent requests for the object are routed directly to its servant.

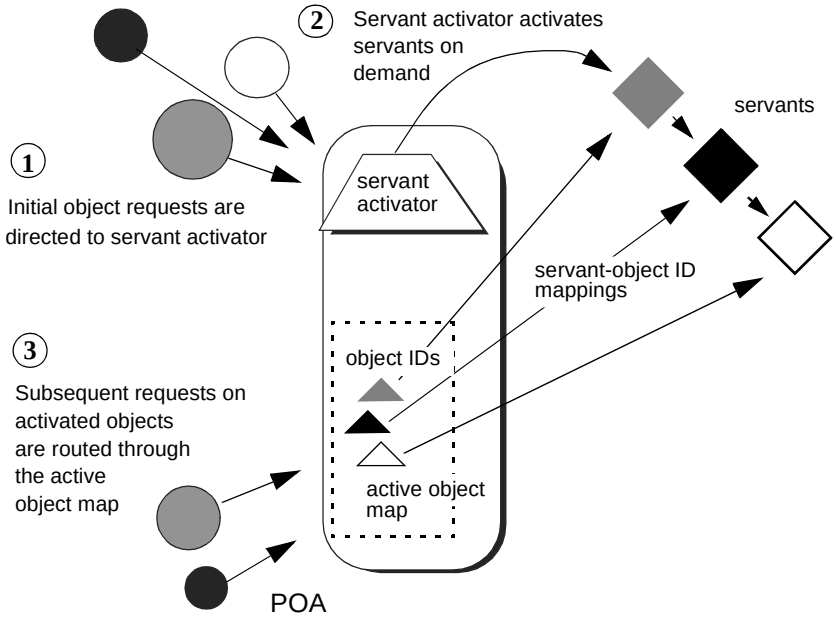


Figure 16: *On the first request on an object, the servant activator returns a servant to the POA, which establishes the mapping in its active object map.*

Servant activators are generally useful when a server can hold all its servants in memory at once, but the servants are slow to initialize, or they are not all needed each time the server runs. In both cases, you can expedite server startup by deferring servant activation until it is actually needed.

ServantActivator interface

The `PortableServer::ServantActivator` interface is defined as follows:

```
interface ServantActivator : ServantManager
{
    Servant
    incarnate(
       in ObjectId oid,
       in POA      adapter
     raises (ForwardRequest);

   void
   etherealize(
      in ObjectId oid,
      in POA      adapter,
      in Servant  serv,
      in boolean  cleanup_in_progress,
      in boolean  remaining_activations
   ;
};
```

A POA can call two methods on its servant activator:

- `incarnate()` is called by the POA when it receives a request for an inactive object, and should return an appropriate servant for the requested object.
- `etherealize()` is called by the POA when an object is deactivated or the POA shuts down. In either case, it allows the application to clean up resources that the servant uses.

Implementing a servant activator

You can implement a servant activator as follows:

Example 15: *Servant activator implementation*

```
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import demos.servant_management.BankDemo.AccountPackage.*;
import demos.servant_management.BankDemo.*;

public class AccountServantActivatorImpl
    extends LocalObject
    implements ServantActivator
{
    // servant activator constructor
    public AccountServantActivatorImpl(
        AccountDatabase account_db,
        org.omg.CORBA.ORB orb)
    {
        m_account_db = account_db;
        m_orb = orb;
    }

1   public Servant incarnate(
        byte[] oid,
        POA adapter) throws ForwardRequest
   {
       String account_id = new String(oid);

2      SingleAccountImpl account =
           new SingleAccountImpl(account_id, m_account_db);
       return account;
    }

    public void etherealize(byte[] oid,
        POA adapter,
        Servant serv,
        boolean cleanup_in_progress,
        boolean remaining_activations)
    { }

    private AccountDatabase m_account_db;
    private org.omg.CORBA.ORB m_orb;
}
```

In this example, the servant activator's constructor takes two arguments that enable interaction between Account objects and persistent account data: an AccountDatabase object, and the application's ORB

Activating objects

`incarnate()` instantiates a servant for a requested object and returns the servant to the POA. The POA registers the servant with the object's ID, thereby activating the object and making it available to process requests on it.

In the implementation shownshown in Example 15, `incarnate()` performs these tasks:

1. Takes the object ID of a request for a `BankDemo::Account` object, and the POA that relayed the request.
2. Instantiates an `SingleAccountImpl` servant, passing account information to the servant's constructor, and returns the servant to the POA.

Deactivating objects

The POA calls `etherealize()` when an object deactivates, either because the object is destroyed or as part of general cleanup when the POA itself deactivates or is destroyed.

Because Java automatically disposes of servants for deactivated objects, the `etherealize()` method is generally used to perform required cleanup or database interaction before objects deactivate. For example, it can check the `cleanup_in_progress` parameter to determine whether etherealization results from POA deactivation or destruction; this lets you differentiate between this and other reasons to etherealize a servant.

Setting deactivation policies

By default, a POA that uses a servant activator lets an object deactivate (and its servant to etherealize) only after all pending requests on that object return. You can modify the way the POA handles incoming requests for a deactivating object by creating an Orbix-proprietary `ObjectDeactivationPolicy` object and attaching it to the POA's `PolicyList` (see "Setting proprietary policies for a POA" on page 199).

Three settings are valid for this Policy object:

DELIVER: (default) The object deactivates only after processing all pending requests, including any requests that arrive while the object is deactivating. This behavior complies with CORBA specifications.

DISCARD: The POA rejects incoming requests with an exception of `TRANSIENT`. Clients should be able to reissue discarded requests.

HOLD: Requests block until the object deactivates. A POA with a `HOLD` policy maintains all requests until the object reactivates. However, this policy can cause deadlock if the object calls back into itself.

Setting a POA's servant activator

The following example shows how you can establish a POA's servant activator in two steps:

Example 16: *Java Setting the POA's Servant Activator*

```
...
AccountDatabase account_database = new AccountDatabase();

1  // instantiate servant activator
org.omg.PortableServer.ServantActivator activator =
   new AccountServantActivatorImpl(account_database, orb);

2  // Associate the activator with the accounts POA
acct_poa.set_servant_manager( activator );
```

1. Instantiate the servant activator.
2. Call `set_servant_manager()` on the target POA and supply the servant activator.

Servant Locators

A server that needs to manage a large number of objects might only require short-term access to them. For example, the operations that are likely to be invoked on most customer bank accounts—such as withdrawals and deposits—are usually infrequent and of short duration. Thus, it is unnecessary to keep account objects active beyond the lifetime of any given request. A POA that services requests like this can use a servant locator, which activates an object for each request, and deactivates it after the request returns.

Required policies

A POA with policies of `USE_SERVANT_MANAGER` and `NON_RETAIN` uses a servant locator as its servant manager. Because the POA lacks an active object map, it directs each object request to the servant locator, which returns a servant to the POA in order to process the request. The POA calls the request operation on the servant; when the operation returns, the POA deactivates the object and returns control to the servant locator. From the POA's perspective, the servant is active only while the request is being processed.

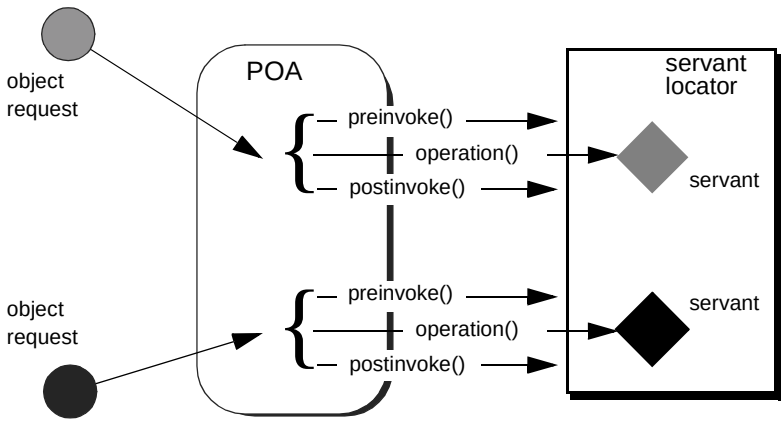


Figure 17: *The POA directs each object request to the servant locator, which returns a servant to the POA to process the request.*

Controlling servant lifespan

An application that uses a servant locator has full control over servant creation and deletion, independently of object activation and deactivation. Your application can assert this control in a number of ways. For example:

- *Servant caching*: A servant locator can manage a cache of servants for applications that have a large number of objects. Because the locator is called for each operation, it can determine which objects are requested most recently or frequently and retain and remove servants accordingly.
- *Application-specific object map*: A servant locator can implement its own object-servant mapping algorithm. For example, a POA's active object map requires a unique servant for each interface. With a servant locator, an application can implement an object map as a simple fixed table that maps multiple objects with different interfaces to the same servant. Objects can be directed to the appropriate servant through an identifier that is embedded in their object IDs. For each incoming request, the servant locator extracts the identifier from the object ID and directs the request to the appropriate servant.

ServantLocator interface

The `PortableServer:ServantLocator` interface is defined as follows:

```
interface ServantLocator : ServantManager
{
    native Cookie;
    Servant
    preinvoke(
        in ObjectId  oid,
        in POA  adapter,
        in CORBA::Identifier operation,
        out Cookie the_cookie
      raises (ForwardRequest);

    void
    postinvoke(
        in ObjectId oid,
        in POA adapter,
        in CORBA::Identifier operation,
        in Cookie the_cookie,
        in Servant the_servant
    ;
};
```

A servant locator processes each object request with a pair of methods, `preinvoke()` and `postinvoke()`:

- `preinvoke()` is called on a POA's servant locator when the POA receives a request for an object. `preinvoke()` returns an appropriate servant for the requested object.
- `postinvoke()` is called on a POA's servant locator to dispose of the servant when processing of the object request is complete. For example, the `postinvoke()` implementation can cache the servant for later reuse.

Implementing a servant locator

The following code implements a servant locator that handles account objects:

Example 17: *Servant locator implementation*

```
package demos.servant_management;

import org.omg.CORBA.*;
import org.omg.PortableServer.POA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.ServantLocatorPackage.*;
import demos.servant_management.BankDemo.AccountPackage.*;
import demos.servant_management.BankDemo.*;

public class AccountServantLocatorImpl
  extends LocalObject
  implements ServantLocator
{
    public AccountServantLocatorImpl(AccountDatabase account_db,
                                     org.omg.CORBA.ORB orb)
    {
        m_account_db = account_db;
        m_orb = orb;
    }
```

Example 17: *Servant locator implementation*

```
    public org.omg.PortableServer.Servant preinvoke(
        byte[] oid,
        POA adapter,
        String operation,
        CookieHolder the_cookie)
    throws ForwardRequest
    {
        String account_id = new String(oid);
        SingleAccountImpl account =
            new SingleAccountImpl(account_id, m_account_db);
     return account;
    }

    public void postinvoke(
        byte[] oid,
        POA adapter,
        String operation,
        java.lang.Object the_cookie,
        org.omg.PortableServer.Servant the_servant)
    {
        if (the_servant instanceof SingleAccountImpl)
        {
            SingleAccountImpl account =
                (SingleAccountImpl)the_servant;
                account.save_all();
        }
    }

    AccountDatabase m_account_db;
    org.omg.CORBA.ORB m_orb;
}
```

Each request is guaranteed a pair of `preinvoke()` and `postinvoke()` calls. This can be especially useful for applications with database transactions. For example, a database server can use a servant locator to direct concurrent operations to the same servant; each database transaction is opened and closed within the `preinvoke()` and `postinvoke()` operations.

The signatures of `preinvoke()` and `postinvoke()` are differentiated from those of `invoke()` and `incarnate()` by two parameters, `the_cookie` and `operation`:

the_cookie lets you explicitly map data between `preinvoke()` and its corresponding `postinvoke()` call. This can be useful in a multi-threaded environment and in transactions where it is important to ensure that a pair of `preinvoke()` and `postinvoke()` calls operate on the same servant. For example, each `preinvoke()` call can set its `the_cookie` parameter to data that identifies its servant; the `postinvoke()` code can then compare that data to its `the_servant` parameter.

operation contains the name of the operation that is invoked on the CORBA object, and thus provides the context of the servant's instantiation. The servant can use this to differentiate between different operations and execute the appropriate code.

Setting a POA's servant locator

You establish a POA's servant locator in two steps, as shown in the following example:

Example 18: *Java Setting a POA's Servant Locator*

```
1  // instantiate a servant locator
   org.omg.PortableServer.ServantLocator locator =
       new AccountServantLocatorImpl(account_database, orb);

2  // Associate the locator with the accounts POA
   acct_poa.set_servant_manager( locator );
```

1. Instantiate the servant locator.
2. Call `set_servant_manager()` on the target POA and supply the servant locator.

Using a Default Servant

If a number of objects share the same interface, a server can most efficiently handle requests on them through a POA that provides a single default servant. This servant processes all requests on a set of objects. A POA with a request processing policy of `USE_DEFAULT_SERVANT` dispatches requests to the default servant when it cannot otherwise find a servant for the requested object. This can occur because the object's ID is not in the active object map, or the POA's servant retention policy is set to `NON_RETAIN`.

For example, all customer account objects in the bank server share the same `BankDemo::Account` interface. Instead of instantiating a new servant for each customer account object as in previous examples, it might be more efficient to create a single servant that processes requests on all accounts.

Obtaining the current object

A default servant must be able to differentiate the objects that it is serving. The `PortableServer::Current` interface offers this capability:

```
module PortableServer
{
    interface Current : CORBA::Current
    {
        exception NoContext{};
        POA get_POA () raises (NoContext);
        ObjectID get_object_id() raises (NoContext);
    };
    ...
}
```

You can call a `PortableServer::Current` operation only in the context of request processing. Thus, each `Bank::Account` operation such as `deposit()` or `balance()` can call `PortableServer::Current::get_object_id()` to obtain the current object's account ID number.

Implementing a default servant

To implement a default servant for account objects, modify the code as follows:

- The `SingleAccountImpl` constructor identifies the ORB instead of an object's account ID.

- Each Account operation calls `resolve_initial_references()` on the ORB to obtain a reference to the `PortableServer::Current` object, and uses this reference to identify the current account object.

So, you might use the following servant code to implement an account object:

Example 19: *Implementation of a default servant*

```
package demos.servant_management;

import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import demos.servant_management.BankDemo.AccountPackage.*;
import demos.servant_management.BankDemo.*;

public class SingleAccountImpl extends AccountPOA
{
    // constructor
    public SingleAccountImpl(ORB orb,
        AccountDatabase account_db)
    {
        m_account_db = account_db;
        m_orb = orb;
     }

     protected void update_balance(float balance)
     {
         m_account_db.write_account(get_account_id(), balance);
     }

    public float balance()
    {
        float balance =
            m_account_db.read_account(get_account_id());
        return balance;
    }
```

Example 19: *Implementation of a default servant*

```
    public void withdraw(float amount) throws InsufficientFunds
    {
        float balance = balance();
        if (amount > balance)
        {
            throw new InsufficientFunds();
        }
        update_balance(get_account_id(), balance - amount);
    }

    private String get_account_id()
    {
        org.omg.CORBA.Object obj =
            m_orb.resolve_initial_references("POACurrent");
        org.omg.PortableServer.Current poa_current =
            org.omg.PortableServer.CurrentHelper.narrow(obj);
        try {
            byte[] account_oid = poa_current.get_object_id();
        } catch (org.omg.PortableServer.Current.NoContext) {
          // ...
        }

        return new String(account_oid);
    }

    private ORB m_orb;
    private AccountDatabase m_account_db;
}
```

In this implementation, the servant constructor takes a single argument, a reference to the ORB. Each method such as `balance()` calls the private helper method `get_account_id()`, which obtains a reference to the current object (`PortableServer::Current`) and gets its object ID. The method converts the object ID to a string and returns with this string.

This implementation assumes that account object IDs are generated from account ID strings. See "Creating Inactive Objects" on page 290 to see how you can create object IDs from a string and use them to generate object references.

Setting a Default Servant

You can establish a POA's default servant by instantiating the desired servant class and supplying it as an argument to `set_servant()`, which you invoke on that POA. The following code fragment from the server's `main()` instantiates servant `def_serv` from servant class `SingleAccountImpl`, and sets this as the default servant for POA `acct_poa`:

```
// Initialize the ORB
CORBA::ORB_var orb = CORBA::ORB_init( argc, argv );

// Instantiate default account object servant
SingleAccountImpl def_serv( orb );
...

// Set default servant for POA
acct_poa->set_servant( &def_serv );
```

```
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
// ...

// Initialize the ORB
public static ORB orb = ORB.init( args, null );

// create account POA with policy of DEFAULT_SERVANT policy
// (not shown)
// ...

// Instantiate default account object servant
try {
    SingleAccountImpl def_serv = new SingleAccountImpl( orb );
    ...

    // Set default servant for POA
    acct_poa.set_servant( def_serv );
}
catch (org.omg.PortableServer.WrongPolicy ex) {
// wrong policy for default servant
}
// ...
```

Creating Inactive Objects

An application that uses a servant manager or default servant typically creates objects independently of the servants that incarnate them. The various implementations shown earlier in this chapter assume that all account objects are available before they are associated with servants in the POA. Thus, the account objects are initially inactive—that is, servants are unavailable to process any requests that are invoked on them.

You can create inactive objects by calling either `create_reference()` or `create_reference_with_id()` on a POA. In the next example, the POA that is to maintain these objects has an ID assignment policy of `USER_ID`; therefore, the server code calls `create_reference_with_id()` to create objects in that POA:

Note: The repetitive mechanism used in this example to create objects is used only for illustrative purposes. A real application would probably use a factory object to create account objects from persistent data.

```
import org.omg.CORBA.*;
import org.omg.PortableServer.*;

public static main( String args[] ) {
    // initialize ORB
    ORB orb = ORB.init( args, null );

    // get object reference to the root POA
   org.omg.CORBA.Object obj =
        orb.resolve_initial_references( "RootPOA" );
    POA root_poa = POAHelper.narrow( obj );

    // set policies for persistent POA that uses servant locator
    Policy[] policies = new Policy[] {
        root_poa.create_lifespan_policy(
            LifespanPolicyValue.PERSISTENT);
        root_poa.create_id_assignment_policy(
            IdAssignmentPolicyValue.USER_ID);
        root_poa.create_servant_retention_policy(
            ServantRetentionPolicyValue.NON_RETAIN);
        root_poa.create_request_processing_policy(
            RequestProcessingPolicyValue.USE_SERVANT_MANAGER);
    };
```

```
    // create the accounts POA
    POA acct_poa = root_poa.create_POA(
        "acct_poa", null, policies );

    // instantiate a servant locator
    org.omg.PortableServer.ServantLocator locator =
        new AccountServantLocatorImpl(orb);

    // Associate the locator with the accounts POA
    acct_poa.set_servant_manager( locator );

    // Set Bank Account interface repository ID
    String repository_id = "IDL:BankDemo/Account:1.0";

    // create account object
    String acct_id =  "112-1110001";
    byte[] acct_oid = acct_id.getBytes();

    org.omg.CORBA.Object acct_obj =
        acct_poa.create_reference_with_id(
        acct_oid, repository_id );

    // Export object reference to Naming Service (not shown)

    // create another account object
    acct_id = "112-1110002";
    acct_oid = acct_id.getBytes();
    acct_obj = acct_poa.create_reference_with_id(
        acct_oid, repository_id);

    // Export object reference to Naming Service (not shown)

    // Repeat for each account object...

    // Start ORB
    orb.run();
    return 0;
}
```

As shown, `main()` executes as follows:

1. Creates all account objects in `acct_poa` without incarnating them.
2. Calls `run()` on the ORB so it starts listening to requests.
3. As the POA receives requests for objects, it passes them on to the servant locator. The servant locator instantiates a servant to process each request.

4. After the request returns from processing, the servant locator destroys its servant.

Exceptions

Implementations of IDL operations and attributes throw exceptions to indicate when a processing error occurs.

An IDL operation can throw two types of exceptions:

- *User-defined exceptions* are defined explicitly in your IDL definitions.
- *System exceptions* are predefined exceptions that all operations can throw.

While IDL operations can throw user-defined and system exceptions, accessor methods for IDL attributes can only throw system-defined exceptions.

Example IDL

This chapter shows how to throw and catch both types of exceptions. The `Bank` interface is modified to include two user-defined exceptions:

AccountNotFound is defined by `find_account()`.

AccountAlreadyExists is defined by `create_account()`.

The account_id member in both exceptions indicates an invalid account ID:

```
module BankDemo
{
    ...
    interface Bank {
        exception AccountAlreadyExists { AccountId account_id; };
        exception AccountNotFound      { AccountId account_id; };

        Account find_account(in AccountId account_id)
            raises(AccountNotFound);

        Account create_account(
            in AccountId account_id,
            in CashAmount initial_balance
        ) raises (AccountAlreadyExists);
    };
};
```

In this chapter

This chapter contains the following sections:

Exception Code Mapping

All CORBA exceptions ultimately derive from `java.lang.Exception`, as shown in Figure 18, and can be instantiated and manipulated like any Java exception object:

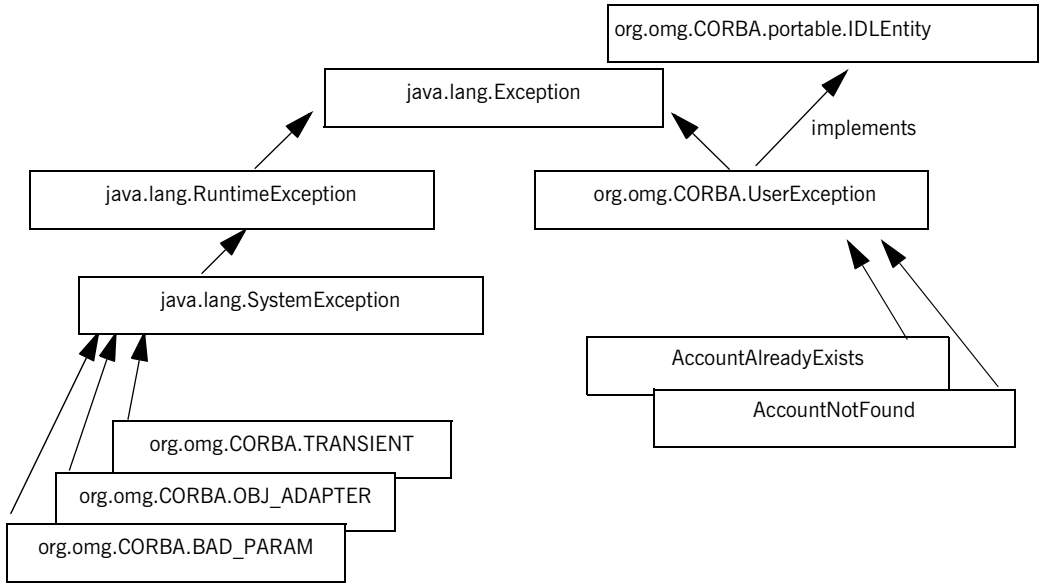


Figure 18: *The Java mapping arranges exceptions into a hierarchy*

Subclasses

CORBA exceptions are subdivided into two subclasses:

- System exceptions are subclasses of `org.omg.CORBA.SystemException`. All system exceptions are defined by the OMG.
- User exceptions are described in the IDL that you write; these are subclasses of `org.omg.CORBA.UserException`. The IDL compiler places user exceptions into Java packages that are scoped to the interface or module in which the exception was defined.

Given this hierarchy, you can catch all CORBA exceptions in a single catch handler. Alternatively, you can catch system and user exceptions separately, or handle specific exceptions individually.

User-Defined Exceptions

Operations are defined to raise one or more user exceptions to indicate application-specific error conditions. An exception definition can contain multiple data members to convey specific information about the error, if desired. For example, you might include a graphic image in the exception data in order to display an error icon.

Exception design guidelines

When you define exceptions, be sure to follow these guidelines:

Exceptions are thrown only for exceptional conditions. Do not throw exceptions for expected outcomes. For example, a database lookup operation should not throw an exception if a lookup does not locate anything; it is normal for clients to occasionally look for things that are not there. It is harder for the caller to deal with exceptions than return values, because exceptions break the normal flow of control. Do not force the caller to handle an exception when a return value is sufficient.

Exceptions carry complete information. Ensure that exceptions carry all the data the caller requires to handle an error. If an exception carries insufficient information, the caller must make a second call to retrieve the missing information. However, if the first call fails, it is likely that subsequent calls will also fail.

Exceptions only carry useful information. Do not add exception members that are irrelevant to the caller.

Exceptions carry precise information Do not lump multiple error conditions into a single exception type. Instead, use a different exception for each semantic error condition; otherwise, the caller cannot distinguish between different causes for an error.

Java mapping for user exceptions

If an exception is defined within an interface, then its Java class name is defined within a package that corresponds to the IDL interface's name. Thus, exception class `AccountAlreadyExists` is defined within package `BankDemo.BankPackage`:

```
package BankDemo.BankPackage;
public final class AccountAlreadyExists
    extends org.omg.CORBA.UserException
{
    public java.lang.String account_id;

    public AccountAlreadyExists() {
        super(AccountAlreadyExistsHelper.id());
    }

    public AccountAlreadyExists(
        java.lang.String account_id
    )
    {
        super(AccountAlreadyExistsHelper.id());
        this.account_id = account_id;
    }

    public AccountAlreadyExists(
        java.lang.String _reason,
        java.lang.String account_id
    )
    {
        super(AccountAlreadyExistsHelper.id() + " " + _reason);
        this.account_id = account_id;
    }
}
```

Constructors

Three constructors are provided:

- The default constructor takes no arguments.
- The user-defined constructor takes an argument for each exception member—in this case, `account_id`.
- The full constructor contains an additional `reason` parameter that is concatenated to the ID before calling the superclass constructor.

Handling Exceptions

Client code uses standard `try` and `catch` blocks to isolate processing logic from exception handling code. You can associate multiple `catch` blocks with each `try` block. You should write the code so that handling for specific exceptions takes precedence over handling for other unspecified exceptions.

Handling User Exceptions

If an operation might throw a user exception, its caller should be prepared to handle that exception with an appropriate `catch` clause.

Example 20 shows how you might program a client to catch exceptions. In it, the handler for the `AccountAlreadyExists` exception outputs an error message and exits the program.

Example 20: *Programming a client to catch user exceptions*

```
  protected void do_create()  // create bank account
  {
    try {
      System.out.println("Enter account name :");
      String name = m_input.readLine();
      System.out.println("Enter account starting balance :");
      String balance = m_input.readLine();
      Float balance_converter = new Float(balance);
      float float_balance = balance_converter.floatValue();
      System.out.println("Calling create account with " +
                         float_balance);
      Account account = m_bank.create_account(
         name, float_balance);
      AccountMenu sub_menu = new AccountMenu(account);
      sub_menu.run();
    }
    catch (
       BankDemo.BankPackage.AccountAlreadyExists already_exists)
    {
      System.err.println("This account already exists.");
      return;
    }
    catch (java.io.IOException io_exc) {
      System.err.println("Bank menu IO exception.");
      return;
    }
}
```

Handling System Exceptions

A client often provides a handler for a limited set of anticipated system exceptions. It also must provide a way to handle all other unanticipated system exceptions that might occur.

Precedence of exception handlers

The Java runtime first tries to match an exception to a catch block that specifies that exception; otherwise it matches the exception's superclass. Because all CORBA exceptions are derived from `java.lang.Exception`, catch blocks with specific exception handling must precede more general catch blocks.

The following client code specifically tests for a `COMM_FAILURE` exception; it can also handle any other system and I/O exceptions:

Example 21: *Handling system exception* `COMM_FAILURE`

```
public void run() {
    if (m_bank == null) {
        System.err.println(
            "Cannot proceed bank reference is null.");
      return;
    }
    else {
        for (;;) {
            System.err.println("");
            System.err.println("0 - quit");
            System.err.println("1 - create account");
            System.err.println("2 - find account");
            System.err.println("Selection [0-2] :");
```

Example 21: *Handling system exception* `COMM_FAILURE`

```
            try {
                String user_selection = m_input.readLine();
                System.out.println(
                    "You choose [" + user_selection + "]");

                if (user_selection.equals("0")) {
                    return;
                }
                else {
                    if (user_selection.equals("1")) {
                        do_create();
                    }
                    else {
                        if (user_selection.equals("2")) {
                            do_find();
                        }
                    }
                }
            }
            catch (org.omg.CORBA.COMM_FAILURE com) {
                System.err.println(
                    "Communication failure exception"+com);
                return;
            }

            catch (org.omg.CORBA.SystemException sys_exc) {
                System.err.println(
                    "System exception in bank menu"+sys_exc);
                return;
            }
            catch (java.io.IOException io_exc) {
                System.err.println("IO exception in bank menu");
                return;
            }
        }
    }
}
```

Evaluating System Exceptions

Each system exception has two members that let a client evaluate the status of an invocation:

```
abstract public class SystemException extends
    java.lang.RuntimeException {

    public int minor;
    public CompletionStatus completed;
    ...
}
```

completed is set to an integer value that indicates how far the operation or attribute call progressed. You can obtain this value by calling `org.omg.CORBA.CompletionStatus.value()` on it.

minor offers more detail about the particular system exception that was thrown.

Obtaining invocation completion status

Each standard exception includes a `completion_status` code that takes one of the following integer values:

COMPLETED_NO: The system exception was thrown before the operation or attribute call began to execute.

COMPLETED_YES: The system exception was thrown after the operation or attribute call completed execution.

COMPLETED_MAYBE: It is uncertain whether or not the operation or attribute call started to execute, and if so, whether execution completed. For example, the status is `COMPLETED_MAYBE` if a client's host receives no indication of success or failure after transmitting a request to a target object on another host.

Evaluating minor codes

`minor()` returns an IDL `unsigned long` that offers more detail about the particular system exception thrown. For example, if a client catches a `COMM_FAILURE` system exception, it can access the system exception's minor field to determine why this occurred

All standard exceptions have an associated minor code that provides more specific information about the exception in question. Given these minor codes, the ORB is not required to maintain an exhaustive list of all possible exceptions that might arise at runtime.

Minor exception codes are defined as an unsigned long that contains two components:

- 20-bit vendor minor code ID (VMCID)
- Minor code that occupies the 12 low order bits

All minor codes are based on the IONA vendor minor code ID (`IONA_VMCID`), which is `0x49540000`. The space reserved to IONA ends at `0x49540FFF`.

The VMCID assigned to OMG standard exceptions is `0x4f4d000`. You can obtain the minor code value for any exception by OR'ing the VMCID with the minor code for the exception in question. All minor code definitions are associated with readable strings.

Subsystem minor codes

Orbix defines minor codes within each subsystem. When an exception is thrown, the current subsystem associates the exception with a valid minor code that maps to a unique error condition. Table 12 lists Orbix subsystems and base values for their minor codes:

Table 12: *Base minor code values for Orbix subsystems*

Subsystem	Logging ID	Minor Code ID
IT_SOAP	IT_SOAP	IONA_VMCID + 0x080
IT_Core	IT_CORE	IONA_VMCID + 0x100
IT_CONFIG_REP	IT_CONFIG_REP	IONA_VMCID + 0x140
IT_SOAP_Profile	IT_SOAP_PROFILE	IONA_VMCID + 0x180
IT_GIOP	IT_GIOP	IONA_VMCID + 0x200
Thread/Synch Package	IT_TS	IONA_VMCID + 0x240
IT_IIOP	IT_IIOP	IONA_VMCID + 0x300
IT_PSS_ODBC	IT_PSS_ODBC	IONA_VMCID + 0x340
IT_WSDL	IT_WSDL	IONA_VMCID + 0x380
IT_IIOP_PROFILE	IT_IIOP_PROFILE	IONA_VMCID + 0x400

Table 12: *Base minor code values for Orbix subsystems*

Subsystem	Logging ID	Minor Code ID
IT_ATLI_IOP	none	IONA_VMCID + 0x440
IT_ATLI_TCP	IT_ATLI_TCP	IONA_VMCID + 0x480
IT_POA	IT_POA	IONA_VMCID + 0x500
IT_PortableInterceptor	IT_PORTABLE_INTERCEPTOR	IONA_VMCID + 0x540
IT_OTS_TM	IT_OTS_TM	IONA_VMCID + 0x580
IT_PSS_R	IT_PSS_R	IONA_VMCID + 0x600
IT_XA	IT_XA	IONA_VMCID + 0x640
IT_OTS_Encina	IT_OTS_ENCINA	IONA_VMCID + 0x680
IT_PSS_DB	IT_PSS_DB	IONA_VMCID + 0x700
iPAS subsystems	IT_iPAS_*	IONA_VMCID + 0x740
IT_SHMIOP	IT_SHM_IOP	IONA_VMCID + 0x780
IT_PSS	IT_PSS	IONA_VMCID + 0x800
IT_NOTIFICATION	IT_NOTIFICATION	IONA_VMCID + 0x840
IT_ATLI_SHM	IT_ATLI_SHM	IONA_VMCID + 0x880
IT_OTS	IT_OTS	IONA_VMCID + 0x900
IT_TLS	IT_TLS	IONA_VMCID + 0x940
IT_ATLI_MULTICAST	IT_ATLI_MULTICAST	IONA_VMCID + 0x980
IT_OTS_Lite	IT_OTS_LITE	IONA_VMCID + 0xA00
IT_IIOP_TLS	IT_IIOP_TLS	IONA_VMCID + 0xA40
IT_OPAL	IT_OPAL	IONA_VMCID + 0xA80
IT_LOCATOR	IT_LOCATOR	IONA_VMCID + 0xB00
IT_NodeDaemon	IT_NODE_DAEMON	IONA_VMCID + 0xB40
IT_EGMIOP_Component	IT_EGMIOP_COMPONENT	IONA_VMCID + 0xB80
IT_POA_LOCATOR	IT_POA_LOCATOR	IONA_VMCID + 0xC00

Table 12: *Base minor code values for Orbix subsystems*

Subsystem	Logging ID	Minor Code ID
IT_KDM	IT_KDM	IONA_VMCID + 0xC40
IT_EGMIOP	IT_EGMIOP	IONA_VMCID + 0xC80
IT_ACTIVATOR	IT_ACTIVATOR	IONA_VMCID + 0xD00
IT_Daemon	IT_DAEMON	IONA_VMCID + 0xE00
IT_JTA	IT_JTA	IONA_VMCID + 0xE40
IT_NAMING	IT_NAMING	IONA_VMCID + 0xF00
IT_COBOL_PLI	IT_COBOL_PLI	IONA_VMCID + 0xF40
IT_MVS	IT_MVS	IONA_VMCID + 0xF80

For example, the locator subsystem defines a number of minor codes for the BAD_PARAM standard exception. These distinguish among the various conditions under which the locator might throw the BAD_PARAM exception.

Definitions for all subsystem minor codes can be found in the directory asp/*version*/xml/minor_codes.

Note: OMG minor code constants are Orbix-specific mappings to minor codes that are set by the OMG. If you define minor codes for your own application, make sure that they do not overlap the ranges that are reserved for IONA-defined minor codes.

Displaying minor code strings

In order to provide user-readable output for minor codes, Orbix provides helper class com.iona.corba.util.SystemExceptionDisplayHelper. The following catch statement shows how a program typically uses this class:

```
// ...
catch (SystemException ex) {
   System.err.println("Caught exception: " +
   SystemExceptionDisplayHelper.toString(ex));
}
```

This yields output such as the following:

```
Caught exception: org.omg.CORBA.INITIALIZE
minor_code:1230242048 completed:No (IT_Core:ERROR_IN_DOMAIN)
```

Throwing Exceptions

Client code uses standard Java syntax to initialize and throw both user-defined and system exceptions.

This section modifies `BankImpl.create_account()` to throw an exception. You can implement `create_account()` as follows:

Example 22: *Throwing an exception*

```
//  create a new account given an id and initial balance
//  throw AccountAlreadyExists if account already in database

public Account create_account(
    String account_id, float initial_balance)
    throws AccountAlreadyExists
{
    System.out.println(
        "Creating an account with account id of ["
        + account_id + "].");
    if (!(m_account_db.create_account( account_id,
                                       initial_balance))) {
        throw new AccountAlreadyExists();
    }
    return create_account_ref(account_id);
}
```

Throwing System Exceptions

Occasionally, a server program might need to throw a system exception. Specific system exceptions such as `COMM_FAILURE` inherit the `SystemException` constructor:

```
abstract public class
    SystemException extends java.lang.RuntimeException {
        public int minor;
        public CompletionStatus completed;

        // constructor
        protected SystemException(String reason,
            int minor,
            CompletionStatus completed) {
                super(reason);
                this.minor = minor;
                this.completed = completed;
             }
}
final public class
COMM_FAILURE extends org.omg.CORBA.SystemException {
    ...
    public COMM_FAILURE(
        int minor, CompletionStatus completed) { ... }
}
```

The following code uses this constructor to throw a `COMM_FAILURE` exception with minor code `SOCKET_WRITE_FAILED` and completion status `COMPLETED_NO`:

```
// initiate a write for the message
//
try {
    m_connection.write(
        message_buffer,  num_bytes_to_write, offset, timeout);
}
catch (Exception ex) {
    // write failed
    System.out.println("exception occurred during write: " + ex);

    // synchronous write failed
    //
    throw new COMM_FAILURE(
        SOCKET_WRITE_FAILED.value,      // minor code
        CompletionStatus.COMPLETED_NO);
}
```

CHAPTER 13

Using Type Codes

Orbix uses type codes to describe IDL types. The IDL pseudo interface CORBA::TypeCode *lets you describe and manipulate type code values.*

Type codes are essential for the DII and DSI, to specify argument types. The interface repository also relies on type codes to describe types in IDL declarations. In general, type codes figure importantly in any application that handles CORBA::Any data types.

In this chapter

This chapter contains the following sections:

Type Code Components

Type codes are encapsulated in `CORBA::TypeCode` pseudo objects. Each `TypeCode` has two components:

kind: A `CORBA::TCKind` enumerator that associates the type code with an IDL type. For example, enumerators `tk_short`, `tk_boolean`, and `tk_sequence` correspond to IDL types `short`, `boolean`, and `sequence`, respectively.

description: One or more parameters that supply information related to the type code's kind. The number and contents of parameters varies according to the type code.

- The type code description for IDL type `fixed<5,3>` contains two parameters, which specify the number of digits and the scale.
- The type code description for a `string` or `wstring` contains a single parameter that specifies the string's bound, if any.
- Type codes for primitive types require no description, and so have no parameters associated with them—for example, `tk_short` and `tk_long`.

TCKind enumerators

The `CORBA::TCKind` enumeration defines all built-in IDL types:

```
// In module CORBA
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, tk_alias,
    tk_except, tk_longlong, tk_ulonglong, tk_longdouble,
   tk_wchar,
    tk_wstring, tk_fixed, tk_value, tk_value_box, tk_native,
    tk_abstract_interface
};
```

Most of these are self-explanatory—for example, a type code with a `TCKind` of `tk_boolean` describes the IDL type boolean. Some, however, have no direct association with an IDL type:

tk_alias describes an IDL type definition such as `typedef string`.

tk_null describes an empty value condition. For example, if you construct an Any with the default constructor, the Any's type code is initially set to `tk_null`.

tk_Principal is deprecated for applications that are compliant with CORBA 2.3 and later; retained for backward compatibility with earlier applications that use the BOA.

tk_TypeCode describes another type code value.

tk_value describes a value type.

tk_value_box describes a value box type.

tk_void is used by the interface repository to describe an operation that returns no value.

Table 13 shows type code parameters. The table omits type codes with an empty parameter list.

Table 13: *Type Codes and Parameters*

TCKind	Parameters
tk_abstract_interface	*repository-id, name*
tk_alias	*repository-id, name, type-code*
tk_array	*type-code, length...*
tk_enum	*repository-id, name, { member-name }...*
tk_except	*repository-id, name,* *{ member-name, member-type-code }...*
tk_fixed	*digits, scale*
tk_native	*repository-id, name*
tk_objref	*repository-id, name*
tk_sequence	*element-type-code, max-length*[a]

Table 13: *Type Codes and Parameters*

TCKind	Parameters
`tk_string` `tk_wstring`	*max-length*[a]
`tk_struct`	*repository-id, name,* *{ member-name, member-type-code }...*
`tk_union`	*repository-id, name, switch-type-code, default-index,* *{ member-label, member-name, member-type-code }...*
`tk_value`	*repository-id, name, type-modifier, type-code,* *{ member-name, member-type-code, visibility }...*
`tk_value_box`	*repository-id, name,* *{ member-name, member-type-code} ...*

a. For unbounded sequences, strings, and wstrings, this value is 0

Type Code Operations

The `CORBA::TypeCode` interface provides a number of operations that you can use to evaluate and compare `TypeCode` objects. These operations can be divided into two categories:

- General type code operations that can be invoked on all `TypeCode` objects.
- Type-specific operations that are associated with `TypeCode` objects of a specific `TCKind`, and raise a `BadKind` exception if invoked on the wrong type code.

General Type Code Operations

The following operations are valid for all TypeCode objects:

- equal(), equivalent()
- get_compact_typecode()
- kind()

equal(), equivalent()

```
boolean equal( in TypeCode tc );
boolean equivalent( in TypeCode tc );
```

`equal()` and `equivalent()` let you evaluate a type code for equality with the specified type code, returning true if they are the same:

equal() requires that the two type codes be identical in their `TCKind` and all parameters—member names, type names, repository IDs, and aliases.

equivalent() resolves an aliased type code (`TCKind = tk_alias`) to its base, or unaliased type code before it compares the two type codes' `TCKind` parameters. This also applies to aliased type codes of members that are defined for type codes such as `tk_struct`.

For both operations, the following parameters are always significant and must be the same to return true:

- Number of members for `TCKinds` of `tk_enum`, `tk_excep`, `tk_struct`, and `tk_union`.
- Digits and scale for `tk_fixed` type codes.
- The value of the bound for type codes that have a bound parameter—`tk_array`, `tk_sequence`, `tk_string` and `tk_wstring`.
- Default index for `tk_union` type codes.
- Member labels for `tk_union` type codes. Union members must also be defined in the same order.

You must use `equal()` and `equivalent()` to evaluate a type code. For example, the following code is illegal:

```
org.omg.CORBA.Any another_any =
    org.omg.CORBA.ORB.init().create_any();
another_any.insert_string("Hello world");

org.omg.CORBA.TypeCode tc_string =
    org.omg.CORBA.ORB.init().create_string_tc(0);
org.omg.CORBA.TypeCode t = another_any.type();

if (t==tc_string) { ... }  // ERROR!  Bad code.
```

You can correct this code as follows:

```
org.omg.CORBA.Any another_any =
    org.omg.CORBA.ORB.init().create_any();
another_any.insert_string("Hello world");

org.omg.CORBA.TypeCode tc_string =
    org.omg.CORBA.ORB.init().create_string_tc(0);
org.omg.CORBA.TypeCode t = another_any.type();

//Test for exact equality
if (t.equal(tc_string) ) { ... }

//Test for equality, ignoring aliases
if (t.equivalent(tc_string) ) { ... }
```

get_compact_typecode()

```
TypeCode get_compact_typecode();
```

`get_compact_typecode()` removes type and member names from a type code. This operation is generally useful only to applications that must minimize the size of type codes that are sent over the wire.

kind()

```
TCKind kind();
```

`kind()` returns the `TCKind` of the target type code. You can call `kind()` on a `TypeCode` to determine what other operations can be called for further processing:

```
org.omg.CORBA.Any another_any = null;
// Create and initialize 'another_any' (not shown)...

org.omg.CORBA.TypeCode t = another_any.type();

if (t.kind()==org.omg.CORBA.TCKind.tk_short) {
    //...
}
else if (t.kind()==org.omg.CORBA.TCKind.tk_long) {
    //...
}
```

Type-Specific Operations

Table 14 shows operations that can be invoked only on certain type codes. In general, each operation gets information about a specific type-code parameter. If invoked on the wrong type code, these operations raise an exception of `BadKind`.

Table 14: *Type-Specific Operations*

TCKind	Operations

`tk_alias`	`id()`
	`name()`
	`content_type()`

`tk_array`	`length()` `content_type()`

`tk_enum`	`id()`
	`name()`
	`member_count()`
	`member_name()`

`tk_except`	`id()`
	`name()`
	`member_count()`
	`member_name()`
	`member_type()`

Table 14: *Type-Specific Operations*

TCKind	Operations
`tk_fixed`	`fixed_digits()` `fixed_scale()`
`tk_native`	`id()` `name()`
`tk_objref`	`id()` `name()`
`tk_sequence`	`length()` `content_type()`
`tk_string` `tk_wstring`	`length()`

`tk_struct`	`id()`
	`name()`
	`member_count()`
	`member_name()`
	`member_type()`

`tk_union`	`id()`
	`name()`
	`member_count()`
	`member_name()`
	`member_label()`
	`discriminator_type()`
	`default_index()`

`tk_value`	`id()`
	`name()`
	`member_count()`
	`member_name()`
	`member_type()`
	`type_modifier()`
	`concerte_base_type()`
	`member_visibility()`

`tk_value_box`	`id()`
	`name()`
	`member_name()`

Table 15 briefly describes the information that you can access through type code-specific operations. For detailed information about these operations, see the *CORBA Programmer's Reference*.

Table 15: *Information Obtained by Type-Specific Operations*

Operation	Returns:
`concrete_base_type()`	Type code of the concrete base for the target type code; applies only to value types.
`content_type()`	For aliases, the original type. For sequences and arrays, the specified member's type.
`default_index()`	Index to a union's default member. If no default is specified, the operation returns `-1`.
`discriminator_type()`	Type code of the union's discriminator.
`fixed_digits()`	Number of digits in a fixed-point type code.
`fixed_scale()`	Scale of a fixed-point type code.
`id()`	Type code's repository ID.
`length()`	Value of the bound for a type code with `TCKind` of `tk_string`, `tk_wstring`, `tk_sequence`, or `tk_array`.
`member_count()`	Number of members in the type code.
`member_label()`	An `Any` value that contains the value of the union case label for the specified member.
`member_name()`	Name of the specified member. If the supplied index is out of bounds (greater than the number of members), the function raises the `TypeCode::Bounds` exception.
`member_type()`	Type code of the specified member. If the supplied index is out of bounds (greater than the number of members), the function raises the `TypeCode::Bounds` exception.

Table 15: *Information Obtained by Type-Specific Operations*

Operation	Returns:
`member_visibility()`	The `Visibility` (`PRIVATE_MEMBER` or `PUBLIC_MEMBER`) of the specified member.
`name()`	Type code's user-assigned unscoped name.
`type_modifier()`	Value modifier that applies to the value type that the target type code represents.

Type Codes for Basic Types

The Java mapping provides the `get_primitive_tc()` method for generating basic type codes:

```
public org.omg.CORBA.TypeCode
   org.omg.CORBA.ORB.get_primitive_tc(
    org.omg.CORBA.TCKind tcKind
);
```

`get_primitive_tc()` takes one of the basic `TCKind` enumerated constants as an argument and returns a reference to the corresponding basic type code.

For example, the following code obtains a reference to a `boolean` type code:

```
import org.omg.CORBA.*;

TypeCode tc_bool =
   ORB.init().get_primitive_tc(TCKind.tk_boolean);
```

Type Codes for User-Defined Types

For each user-defined type in your IDL, the IDL compiler generates a corresponding *user_defined_type*`Helper` class. A type code for *user_defined_type* is returned by the following method:

```
public static org.omg.CORBA.TypeCode
   user_defined_typeHelper.type();
```

This method is useful when testing the contents of an any (see page 325).

For example, given the following IDL:

```
interface Interesting {
    typedef long longType;
    struct Useful
    {
        longType l;
    };
};
```

type codes for the user-defined types can be obtained as follows:

```
import org.omg.CORBA.*;

TypeCode tc_Interesting = InterestingHelper.type();
TypeCode tc_longType = InterestingPackage.longTypeHelper.type();
TypeCode tc_Useful   = InterestingPackage.UsefulHelper.type();
```

Using the Any Data Type

IDL's any type lets you specify values that can express any IDL type.

This allows a program to handle values whose types are not known at compile time. The any type is most often used in code that uses the interface repository or the dynamic invocation interface (DII).

IDL-Java mapping

The IDL any type maps to the Java `org.omg.CORBA.Any` class. Conceptually, this class contains the following two instance variables:

type is a `TypeCode` object that provides full type information for the value contained in the any. The `Any` class provides a `type()` method to return the `TypeCode` object.

value is the internal representation used to store `Any` values and is accessible via standard insertion and extraction methods.

For example, the following interface, AnyDemo, contains an operation that defines an any parameter:

```
// IDL
interface AnyDemo {
    // Takes in any type that can be specified in IDL
    void passSomethingIn (in any any_type_parameter);

    // Passes out any type specified in IDL
    any getSomethingBack();

    ...
};
```

Given this interface, a client that calls passSomethingIn()constructs an any that specifies the desired IDL type and value, and supplies this as an argument to the call. On the server side, the AnyDemo implementation that processes this call can determine the type of value the any stores and extract its value.

In this chapter

This chapter covers the following topics:

Constructing an Any Object

You must use the `ORB` class (in package `org.omg.CORBA`) to construct `Any` objects. This is illustrated by the following example:

```
import org.omg.CORBA.*;

Any a = ORB.init().create_any();
```

Inserting Basic Types

The Java class Any contains a number of insertion methods that you can use to insert any of the pre-defined IDL types into an Any object. The insertion methods for basic types are:

```
// Class 'org.omg.CORBA.Any' method signatures

public void insert_short(short s);
public void insert_long(int i);
public void insert_longlong(long l);
public void insert_ushort(short s);
public void insert_ulong(int i);
public void insert_ulonglong(long l);
public void insert_float(float f);
public void insert_double(double d);
public void insert_boolean(boolean b);
public void insert_char(char c);
public void insert_wchar(char c);
public void insert_octet(byte b);
public void insert_any(Any a);
public void insert_Object(Object o);

// throw exception when type code inconsistent with value
public void insert_Object(Object o, TypeCode t)
        throws org.omg.CORBA.MARSHAL;
public void insert_string(String s)
        throws org.omg.CORBA.DATA_CONVERSION,
               org.omg.CORBA.MARSHAL;
public void insert_wstring(String s)
        throws org.omg.CORBA.MARSHAL;
public void insert_TypeCode(TypeCode t);
public void insert_fixed(java.math.BigDecimal value);
public void insert_fixed(
               java.math.BigDecimal   value,
               org.omg.CORBA.TypeCode type
        )
        throws org.omg.CORBA.BAD_INV_ORDER;
```

Assume that a client programmer wishes to pass an any containing an IDL short as the parameter to the AnyDemo::passSomethingIn() operation. The following insertion method, which is a member of class Any, can be used:

```
public void insert_short(short s);
```

The client programmer can then write the following code:

```
// Client.java

import org.omg.CORBA.*;
AnyDemo anyDemoObj = null;
Any a = ORB.init().create_any();
short toPass = 26;

try {
    anyDemoObj = // initialize the object reference...

    a.insert_short(toPass);

    anyDemoObj.passSomethingIn(a);
}
catch (SystemException se) {
    ...
}
```

Inserting User-Defined Types

Helper classes for user-defined types provide `insert()` methods to support the insertion of user-defined types into an any. The general form of the signature for `insert()` is:

```
public void user-defined-typeHelper.insert(
    org.omg.CORBA.Any a,
    user_defined_type value
);
```

user-defined-type is the Java type mapped from the user-defined IDL type.

For example, consider the following Foo struct defined in IDL:

```
// IDL
struct Foo {
    string bar;
    float number;
};
```

To pass the Foo struct inside an any parameter, the client programmer can write the following:

Example 23: *Inserting a short into an Any*

```
// Client.java,

import org.omg.CORBA.*;
AnyDemo anyDemoObj = null;
Any a = ORB.init().create_any();

// Initialize the 'Foo' struct
Foo toPass = new Foo();
toPass.bar = "Bar";
toPass.number = (float) 34.5;
```

Example 23: *Inserting a short into an Any*

```
try {
    anyDemoObj = // initialize the object reference...

    FooHelper.insert(a, toPass);

    anyDemoObj.passSomethingIn(a);
}
catch (SystemException se) {
    ...
}
```

Type safety

These insertion methods provide a type-safe mechanism for insertion into an any. Both the type and value of the Any are assigned at insertion. If an attempt is made to insert a value that has no corresponding IDL type, it results in a compile-time error.

Extracting Basic Types

The Any Java class contains a number of methods for extracting pre-defined IDL types from an Any object. These extraction methods are named extract_long(), extract_ulong(), extract_float(), and so on. Each extraction method simply returns a value of the appropriate type.

For example, the signature of the method to extract an IDL short from an any is:

```
// Defined in class 'org.omg.CORBA.Any'
public short extract_short()
    throws org.omg.CORBA.BAD_OPERATION;
```

The BAD_OPERATION system exception is thrown if the type inside the any does not match the type you are trying to extract.

You can extract a basic type from an any as follows:

Example 24: *Extracting a basic type from an Any*

```
// Client.java
import org.omg.CORBA.*;

AnyDemo anyDemoObj = null;
Any a;
short toReceive;

try {
    anyDemoObj = // initialize the object reference...

    a = anyDemoObj.getSomethingBack();

    // extract a short value
    if ((a.type()).kind() == TCKind.tk_short) {
        toReceive = a.extract_short();
    }
}
catch (org.omg.CORBA.BAD_OPERATION bo) {
    ...
}
catch (SystemException se) {
    ...
}
```

Before extracting the value from an any, you must check its type code with `org.omg.CORBA.Any.type()`. For basic types, it is enough to check the `kind()` field of the type code.

Extracting User-Defined Types

User-defined type helper classes provide `extract()` methods, which support the extraction of user-defined types from an `any`. The general form of the signature for `extract()` is:

```
public user_defined_type user_defined_typeHelper.extract(
    org.omg.CORBA.Any a
)
throws org.omg.CORBA.BAD_OPERATION;
```

`user_defined_type` is the Java type mapped from the user-defined IDL type. The `BAD_OPERATION` system exception is thrown if the type inside the `any` does not match the type you are trying to extract.

For example, consider the following `LongSeq` sequence defined in IDL:

```
// IDL
typedef sequence<long, 10> LongSeq;
```

To extract the `LongSeq` sequence from an `any` parameter, you can write the following:

Example 25: *Extracting a user-defined type from an Any*

```
// Client.java

AnyDemo anyDemoObj = null;
org.omg.CORBA.Any a;
long[] toReceive;

try {
    anyDemoObj = // initialize the object reference...

    a = anyDemoObj.getSomethingBack();

    // extract a sequence of longs
    if ((a.type()).equal(LongSeqHelper.type())) {
        toReceive = LongSeqHelper.extract(a);
    }
}
```

Example 25: *Extracting a user-defined type from an Any*

```
catch (org.omg.CORBA.BAD_OPERATION bo) {
    ...
}
catch (SystemException se) {
    ...
}
```

Orbix does not destroy the value of an any after extraction. You can therefore extract the value of an any more than once.

Inserting and Extracting Bounded String Aliases

Bounded strings are usually given an alias using an IDL `typedef` declaration. For example, consider the following definition of the `BoundedString` IDL type:

```
//IDL
typedef string<128> BoundedString;
```

Inserting a bounded string

When the IDL is compiled, a `BoundedStringHelper` class is generated. You can insert a bounded string of `BoundedString` type into an `any` using the standard approach for user-defined types. For example:

```
import org.omg.CORBA.*;
Any a = ORB.init().create_any();
//...
BoundedStringHelper.insert(a, "Less than 128 characters.");
```

Extracting a bounded string

Extraction is performed in a similar way to other user-defined types. To extract the bounded string alias, you can use the `extract()` method of the `BoundedStringHelper` class. For example:

```
import org.omg.CORBA.*;
Any a = ORB.init().create_any();
//...
if ((a.type()).equal(BoundedStringHelper.type()) ) {
    String s = BoundedStringHelper.extract(a);
}
```

Extracting Object References

You can use two methods to extract object references from an any:

- extract() is defined on the associated Helper class.
- extract_Object() is defined on the Any class.

The examples in the following sections use the following two IDL interfaces, BaseIntf and DerivedIntf:

```
//IDL
interface BaseIntf { };

interface DerivedIntf : BaseIntf { };
```

extract()

DerivedIntfHelper.extract() is used to extract an object reference when the most derived type of the object is DerivedIntf. It follows the usual pattern for extracting user-defined types. For example:

Example 26: *Extracting an object reference*

```
AnyDemo anyDemoObj = null;
org.omg.CORBA.Any a;
DerivedIntf toReceive = null;

try {
    anyDemoObj = // initialize the object reference...

    // 'a' contains a 'DerivedIntf' object reference
    a = anyDemoObj.getSomethingBack();

    // extract a 'DerivedIntf' object reference
    if ((a.type()).kind()==org.omg.CORBA.TCKind.tk_objref) {
        toReceive = DerivedIntfHelper.extract(a);
    }
}
```

Example 26: *Extracting an object reference*

```
catch (org.omg.CORBA.BAD_OPERATION bo) {
    ...
}
catch (SystemException se) {
    ...
}
```

extract_Object()

`Any.extract_Object()` is useful when you need to perform a polymorphic extraction from an any—that is, the any contains a derived object reference type and you want to extract it as a base type.

The following example extracts a `DerivedIntf` object reference as a `BaseIntf` object reference:

Example 27: *Extracting a derived object reference type*

```
// Client.java

AnyDemo anyDemoObj = null;
org.omg.CORBA.Any a;
BaseIntf toReceive = null;

try {
    org.omg.CORBA.Object obj;
    anyDemoObj = // initialize the object reference...

    // 'a' contains a 'DerivedIntf' object reference
    a = anyDemoObj.getSomethingBack();

    // extract a 'DerivedIntf' object reference as a 'BaseIntf'
    if ((a.type()).kind()==org.omg.CORBA.TCKind.tk_objref) {
        obj = a.extract_Object();
        toReceive = BaseIntfHelper.narrow(obj);
    }
}
catch (org.omg.CORBA.BAD_OPERATION bo) {
    ...
}
catch (SystemException se) {
    ...
}
```

The any is extracted to `obj` of type `CORBA.Object` using `Any.extract_Object()`. The `obj` object reference is then narrowed to type `BaseInt`.

The remote `DerivedIntf` object can now be invoked on polymorphically, using the object reference of `BaseIntf` type.

Any as a Parameter or Return Value

The mapping for IDL `any` operation parameters and return values are illustrated by the following IDL operation:

```
// IDL
any op1 (in any a1, out any a2, inout any a3);
```

This IDL operation maps to the following Java method:

```
import org.omg.CORBA.Any;
import org.omg.CORBA.AnyHolder;

public Any op1 (Any a1, AnyHolder a2, AnyHolder a3);
```

Both `inout` and `out` parameters map to type `AnyHolder` as explained in "Holder Class Types" on page 250.

Using DynAny Objects

The DynAny interface allows applications to compose and decompose any type values dynamically. With DynAny, you can compose a value at runtime whose type was unknown when the application was compiled, and transmit that value as an any. Conversely, an application can receive a value of type any from an operation, and interpret its type and extract its value without compile-time knowledge of its IDL type.

Interface hierarchy

The DynAny API consists of nine interfaces. One of these, interface DynAnyFactory, lets you create DynAny objects. The rest of the DynAny API consists of the DynAny interface itself and derived interfaces, as shown in Figure 19.

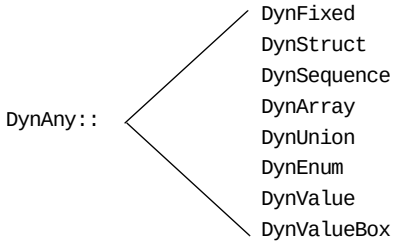


Figure 19: *Interfaces that derive from the DynAny interface*

The derived interfaces correspond to complex, or constructed IDL types such as array and struct. Each of these interfaces contains operations that are specific to the applicable type.

The DynAny interface contains a number of operations that apply to all DynAny objects; it also contains operations that apply to basic IDL types such as long and string.

The DynStruct interface is used for both IDL struct and exception types.

Generic operations

The DynAny interface contains a number of operations that can be invoked on any basic or constructed DynAny object:

```
interface DynAny {
    exception InvalidValue{};
    exception TypeMisMatch {};
    // ...

    void assign(in DynAny dyn_any) raises (TypeMismatch);
    DynAny copy();
    void destroy();

    boolean equal(in DynAny da);

    void from_any(
        in any value) raises(TypeMismatch, InvalidValue);
    any to_any();

    CORBA::TypeCode type();
    // ...
};
```

assign() initializes one DynAny object's value from another. The value must be compatible with the target DynAny's type code; otherwise, the operation raises an exception of TypeMismatch.

copy() creates a DynAny whose value is a deep copy of the source DynAny's value.

destroy() destroys a DynAny and its components.

equal() returns true if the type codes of the two DynAny objects are equivalent and if (recursively) all component DynAny objects have identical values.

from_any() initializes a DynAny object from an existing any object. The source any must contain a value and its type code must be compatible with that of the target DynAny; otherwise, the operation raises an exception of TypeMismatch.

to_any() initializes an any with the DynAny's value and type code.

type() obtains the type code associated with the DynAny object. A DynAny object's type code is set at the time of creation and remains constant during the object's lifetime.

Creating a DynAny

The DynAnyFactory interface provides two creation operations for DynAny objects:

```
module DynamicAny {
  interface DynAny; // Forward declaration

  //...
  interface DynAnyFactory
  {
    exception InconsistentTypeCode {};

    DynAny create_dyn_any(in any value)
        raises (InconsistentTypeCode);
    DynAny create_dyn_any_from_type_code(in CORBA::TypeCode type)
        raises (InconsistentTypeCode);
  };
};
```

Create operations

The create operations return a DynAny object that can be used to manipulate any objects:

create_dyn_any() is a generic create operation that creates a DynAny from an existing any and initializes it from the any's type code and value.

The type of the returned DynAny object depends on the any's type code. For example: if the any contains a struct, create_dyn_any() returns a DynStruct object.

create_dyn_any_from_type_code() creates a DynAny from a type code. The value of the DynAny is initialized to an appropriate default value for the given type code. For example, if the DynAny is initialized from a string type code, the value of the DynAny is initialized to "" (empty string).

Returned types

The type of the returned DynAny object depends on the type code used to initialize it. For example: if a struct type code is passed to create_dyn_any_from_type_code(), a DynStruct object is returned.

If the returned DynAny type is one of the constructed types, such as a DynStruct, you can narrow the returned DynAny before processing it further.

create_dyn_any()

`create_dyn_any()` is typically used when you need to parse an any to analyse its contents. For example, given an any that contains an `enum` type, you can extract its contents as follows:

Example 28: *Creating a DynAny*

```
//Java
import org.omg.CORBA.*;
import org.omg.CORBA.ORBPackage.*;
//...
public void get_any_val(org.omg.CORBA.Any a) {
    org.omg.DynamicAny.DynAnyFactory dyn_fact = null;

    // Get a reference to a 'DynamicAny::DynAnyFactory' object
    try {
1       org.omg.CORBA.Object obj
            = orb.resolve_initial_references("DynAnyFactory");
        dyn_fact
           = org.omg.DynamicAny.DynAnyFactoryHelper.narrow(obj);

        // Get the Any's type code
        org.omg.CORBA.TypeCode tc = a.type();
        if (tc.kind()==TCKind.tk_enum) {
2           org.omg.DynamicAny.DynAny da
                = dyn_fact.create_dyn_any(a);
            org.omg.DynamicAny.DynEnum de
                = org.omg.DynamicAny.DynEnumHelper.narrow(da);
            // ...
3           de.destroy();
        }
        else if (tc.kind()== ... ) {
            //...
        }
    }
    catch (SystemException se) {
        // error: handle exception
    }
    catch (Exception ex) {
        // error: handle exception
    }
}
```

The code executes as follows:

1. To obtain an initial reference to the `DynAnyFactory` object, call `resolve_initial_references("DynAnyFactory")`.

   The `orb` refers to an existing `ORB` object that has been initialized prior to this code fragment.

   The plain `org.omg.CORBA.Object` object reference must be narrowed to the `DynAnyFactory` type before it is used.

2. The `DynAny` created in this step is initialized with the same type and value as the given `CORBA.Any` data type.

   Because the `any` argument of `create_dyn_any()` contains an `enum`, the return type of `create_dyn_any()` is a `DynEnum`. The return value can therefore be narrowed to this type.

3. `destroy()` must be invoked on the `DynAny` object when you are finished with it.

create_dyn_any_from_type_code()

`create_dyn_any_from_type_code()` is typically used to create an any when stub code is not available for the particular type.

For example, consider the IDL `string<128>` bounded string type. In Java there is no `Helper` type available to insert this anonymous bounded string type. You can create an any containing this type as follows:

Example 29: *Inserting an anonymous bounded string.*

```
import org.omg.CORBA.*;
import org.omg.CORBA.ORBPackage.*;
//...
org.omg.DynamicAny.DynAnyFactory dyn_fact = null;

// Get a reference to a 'DynamicAny::DynAnyFactory' object
try {
1       org.omg.CORBA.Object obj
            = orb.resolve_initial_references("DynAnyFactory");
        dyn_fact
            = org.omg.DynamicAny.DynAnyFactoryHelper.narrow(obj);

        // Create type code for an anonymous bounded string type
        int bound = 128;
2       TypeCode tc_v = orb.create_string_tc(bound);

        // Initialize a 'DynAny' containing a bounded string
3       org.omg.DynamicAny.DynAny dyn_bounded_str
            = dyn_fact.create_dyn_any_from_type_code(tc_v);
4       dyn_bounded_str.insert_string("Less than 128 characters.");

        // Convert 'DynAny' to a plain 'any'
5       org.omg.CORBA.Any a = dyn_bounded_str.to_any();
        //...
        // Cleanup 'DynAny'
6       dyn_bounded_str.destroy();
}
catch (SystemException se) {
    // error: handle exception
}
catch (Exception ex) {
    // error: handle exception
}
```

The code can be explained as follows:

1. The initialization service gets an initial reference to the `DynAnyFactory` object by calling `resolve_initial_references("DynAnyFactory")`.

   The `orb` refers to an existing `ORB` object that has been initialized prior to this code fragment.

   The plain `org.omg.CORBA.Object` object reference must be narrowed to the `DynAnyFactory` type before it is used.
2. The `ORB` class supports a complete set of functions for the dynamic creation of type codes. For example, `create_string_tc()` creates bounded or unbounded string type codes. The argument of `create_string_tc()` can be non-zero, to specify the bound of a bounded string, or zero, for unbounded strings.
3. A `DynAny` object, called `dyn_bounded_str`, is created using `create_dyn_any_from_type_code()`. `dyn_bounded_str` is initialized with its type equal to the given bounded string type code, and its value equal to a blank string.
4. The value of `dyn_bounded_str` is set to the given argument of `insert_string()`. Insertion operations of the form `insert_`*`BasicType`* are defined for all basic types, as described in "Accessing basic DynAny values" on page 350.
5. The `dyn_bounded_str` object is converted to a plain `any` that is initialized with the same type and value as the `DynAny`.
6. `destroy()` must be invoked on the `DynAny` object when you are finished with it.

Note: A `DynAny` object's type code is established at its creation and cannot be changed thereafter.

Inserting and Extracting DynAny Values

The interfaces that derive from DynAny such as DynArray and DynStruct handle insertion and extraction of any values for the corresponding IDL types. The DynAny interface contains insertion and extraction operations for all other basic IDL types such as string and long.

Accessing basic DynAny values

The DynAny interface contains two operations for each basic type code, to insert and extract basic DynAny values:+

- An insert operation is used to set the value of the DynAny. The data being inserted must match the DynAny's type code.

  The TypeMismatch exception is raised if the value to insert does not match the DynAny's type code.

  The InvalidValue exception is raised if the value to insert is unacceptable—for example, attempting to insert a bounded string that is longer than the acceptable bound. The InvalidValue exception is also raised if you attempt to insert a value into a DynAny that has components when the current position is equal to -1. See "Iterating Over DynAny Components" on page 356.
- Each extraction operation returns the corresponding IDL type.

  The DynamicAny::DynAny::TypeMismatch exception is raised if the value to extract does not match the DynAny's type code.

  The DynamicAny::DynAny::InvalidValue exception is raised if you attempt to extract a value from a DynAny that has components when the current position is equal to -1. See "Iterating Over DynAny Components" on page 356.

It is generally unnecessary to use a DynAny object in order to access any values, as it is always possible to access these values directly (see page 329 and see page 333). Insertion and extraction operations for basic DynAny types are typically used in code that iterates over components of a constructed DynAny, in order to compose and decompose its values in a uniform way (see page 358).

The IDL for insertion and extraction operations is shown in the following sections.

Insertion Operations

The DynAny interface supports the following insertion operations:

```
void insert_boolean(in boolean value)
    raises (TypeMismatch, InvalidValue);
void insert_octet(in octet value)
    raises (TypeMismatch, InvalidValue);
void insert_char(in char value)
    raises (TypeMismatch, InvalidValue);
void insert_short(in short value)
    raises (TypeMismatch, InvalidValue);
void insert_ushort(in unsigned short value)
    raises (TypeMismatch, InvalidValue);
void insert_long(in long value)
    raises (TypeMismatch, InvalidValue);
void insert_ulong(in unsigned long value)
    raises (TypeMismatch, InvalidValue);
void insert_float(in float value)
    raises (TypeMismatch, InvalidValue);
void insert_double(in double value)
    raises (TypeMismatch, InvalidValue);
void insert_string(in string value)
    raises (TypeMismatch, InvalidValue);
void insert_reference(in Object value)
    raises (TypeMismatch, InvalidValue);
void insert_typecode(in CORBA::TypeCode value)
    raises (TypeMismatch, InvalidValue);
void insert_longlong(in long long value)
    raises (TypeMismatch, InvalidValue);
void insert_ulonglong(in unsigned long long value)
    raises (TypeMismatch, InvalidValue);
void insert_longdouble(in long double value)
    raises (TypeMismatch, InvalidValue);
void insert_wchar(in wchar value)
    raises (TypeMismatch, InvalidValue);
void insert_wstring(in wstring value)
    raises (TypeMismatch, InvalidValue);
void insert_any(in any value)
    raises (TypeMismatch, InvalidValue);
void insert_dyn_any(in DynAny value)
    raises (TypeMismatch, InvalidValue);
void insert_val(in ValueBase value)
    raises (TypeMismatch, InvalidValue);
```

For example, the following code fragment invokes `insert_string()` on a `DynAny` to create an `any` value that contains a string:

Example 30: *Creating an any with insert_string()*

```
import org.omg.CORBA.*;
import org.omg.CORBA.ORBPackage.*;
//...
org.omg.DynamicAny.DynAnyFactory dyn_fact = null;

try {
    // Get a reference to a 'DynamicAny::DynAnyFactory' object
    org.omg.CORBA.Object obj
        = orb.resolve_initial_references("DynAnyFactory");
    dyn_fact
        = org.omg.DynamicAny.DynAnyFactoryHelper.narrow(obj);

    // create DynAny with a string value
    org.omg.DynamicAny.DynAny dyn_a;
    dyn_a = dyn_fact.create_dyn_any_from_type_code(
                orb.get_primitive_tc(TCKind.tk_string));
    dyn_a.insert_string("not to worry!");

    // convert DynAny to any
    org.omg.CORBA.Any a = dyn_a.to_any();
    //...
    // destroy the DynAny
    dyn_a.destroy();
}
catch (SystemException se) {
    // error: handle exception
}
catch (Exception ex) {
    // error: handle exception
}
```

Extraction Operations

The IDL extraction operations supported by the DynAny interface are:

```
boolean            get_boolean()
    raises (TypeMismatch, InvalidValue);
octet              get_octet()
    raises (TypeMismatch, InvalidValue);
char               get_char()
    raises (TypeMismatch, InvalidValue);
short              get_short()
    raises (TypeMismatch, InvalidValue);
unsigned short     get_ushort()
    raises (TypeMismatch, InvalidValue);
long               get_long()
    raises (TypeMismatch, InvalidValue);
unsigned long      get_ulong()
    raises (TypeMismatch, InvalidValue);
float              get_float()
    raises (TypeMismatch, InvalidValue);
double             get_double()
    raises (TypeMismatch, InvalidValue);
string             get_string()
    raises (TypeMismatch, InvalidValue);
Object             get_reference()
    raises (TypeMismatch, InvalidValue);
CORBA::TypeCode    get_typecode()
    raises (TypeMismatch, InvalidValue);
long long          get_longlong()
    raises (TypeMismatch, InvalidValue);
unsigned long long get_ulonglong()
    raises (InvalidValue,TypeMismatch);
long double        get_longdouble()
    raises (TypeMismatch, InvalidValue);
wchar              get_wchar()
    raises (TypeMismatch, InvalidValue);
wstring            get_wstring()
    raises (TypeMismatch, InvalidValue);
any                get_any()
    raises (TypeMismatch, InvalidValue);
DynAny             get_dyn_any()
    raises (TypeMismatch, InvalidValue);
ValueBase          get_val()
    raises (TypeMismatch, InvalidValue);
```

For example, the following code converts a basic any to a DynAny. It then evaluates the DynAny's type code in a switch statement and calls the appropriate get_ operation to obtain its value:

Example 31: *Converting a basic any to a DynAny.*

```
import org.omg.CORBA.*;
import org.omg.CORBA.ORBPackage.*;
//...
org.omg.DynamicAny.DynAnyFactory dyn_fact = null;

try {
    // Get a reference to a 'DynamicAny::DynAnyFactory' object
    org.omg.CORBA.Object obj
        = orb.resolve_initial_references("DynAnyFactory");
    dyn_fact
        = org.omg.DynamicAny.DynAnyFactoryHelper.narrow(obj);

    org.omg.CORBA.Any a = ...; // get Any from somewhere

    // create DynAny from Any
    org.omg.DynamicAny.DynAny dyn_a = dyn_fact.create_dyn_any(a);

    // get DynAny's type code
    TypeCode tcode = dyn_a.type();

    // evaluate type code
    if (tcode.kind()==TCKind.tk_short)
    {
        short s = dyn_a.get_short();
        System.out.println("any contains short value of " + s);
    }
    else if (tcode.kind()==TCKind.tk_long)
    {
        int l = dyn_a.get_long();
        System.out.println("any contains long value of " + l);
    }
    // other cases follow
    //...
```

Example 31: *Converting a basic any to a DynAny.*

```
    dyn_a.destroy();
}
catch (SystemException se) {
    // error: handle exception
}
catch (Exception ex) {
    // error: handle exception
}
```

Iterating Over DynAny Components

Five types of DynAny objects contain components that must be accessed to insert or extract values: DynStruct, DynSequence, DynArray, DynUnion, and DynValue. On creation, a DynAny object holds a current position equal to the offset of its first component. The DynAny interface has five operations that let you manipulate the current position to iterate over the components of a complex DynAny object:

```
module DynamicAny {
    //...
    interface DynAny{
        // ...
        // Iteration operations
        unsigned   long component_count();
        DynAny     current_component() raises (TypeMismatch);
        boolean    seek(in long index);
        boolean    next();
        void       rewind();
    };
};
```

component_count() returns the number of components of a DynAny. For simple types such as long, and for enumerated and fixed-point types, this operation returns 0. For other types, it returns as follows:

- sequence: number of elements in the sequence.
- struct, exception and valuetype: number of members.
- array: number of elements.
- union: 2 if a member is active; otherwise 1.

current_component() returns the DynAny for the current component:

```
DynAny current_component()
```

You can access each of the DynAny's components by invoking this operation in alternation with the next() operation. An invocation of current_component() alone does not advance the current position.

If an invocation of current_component() returns a derived type of DynAny, for example, DynStruct, you can narrow the DynAny to this type.

If you call current_component() on a type that has no components, such as a long, it raises the TypeMismatch exception.

If you call `current_component()` when the current position of the `DynAny` is `-1`, it returns a nil object reference.

next() advances the `DynAny`'s current position to the next component, if there is one:

```
boolean next();
```

The operation returns true if another component is available; otherwise, it returns false. Thus, invoking `next()` on a `DynAny` that represents a basic type always returns false.

seek() advances the current position to the specified component:

```
boolean seek (in long index);
```

Like `next()`, this operation returns true if the specified component is available; otherwise, it returns false.

rewind() resets the current position to the `DynAny` object's first component:

```
void rewind();
```

It is equivalent to calling `seek()` with a zero argument.

Undefined current position

In some circumstances the current position can be undefined. For example, if a `DynSequence` object contains a zero length sequence, both the current component and the value of the `DynAny`'s current position are undefined.

The special value `-1` is used to represent an undefined current position.

When the current position is `-1`, an invocation of `current_component()` yields a nil object reference.

The current position becomes undefined (equal to `-1`) under the following circumstances:

- When the `DynAny` object has no components.

  For example, a `DynAny` containing a zero-length sequence or array would have no components.
- Immediately after `next()` returns false.
- If `seek()` is called with a negative integer argument, or with a positive integer argument greater than the largest valid index.

Accessing Constructed DynAny Values

Each interface that derives from DynAny, such as DynArray and DynStruct, contains its own operations which enable access to values of the following DynAny types:

- DynEnum
- DynStruct
- DynUnion
- DynSequence and DynArray
- DynFixed
- DynValue
- DynValueBox

DynEnum

The DynEnum interface enables access to enumerated any values:

```
module DynamicAny {
    //...
    interface DynEnum : DynAny {
        string get_as_string();
        void set_as_string(in string val) raises(InvalidValue);
        unsigned long get_as_ulong();
        void set_as_ulong(in unsigned long val)
            raises(InvalidValue);
    };
};
```

The DynEnum interface defines the following operations:

get_as_string() and set_as_string() let you access an enumerated value by its IDL string identifier or its ordinal value. For example, given this enumeration:

```
enum Exchange{ NYSE, NASD, AMEX, CHGO, DAX, FTSE };
```

set_as_string("NASD") sets the enum's value as NASD, while you can get its current string value by calling get_as_string().

get_as_ulong() and set_as_ulong() provide access to an enumerated value by its ordinal value.

The following code uses a DynEnum to decompose an any value that contains an enumeration:

Example 32: *Using DynEnum*

```
import org.omg.CORBA.*;
import org.omg.CORBA.ORBPackage.*;
//...

public void extract_any(org.omg.CORBA.Any a){
    org.omg.DynamicAny.DynAnyFactory dyn_fact = null;

    try {
        // Get a reference to a 'DynamicAny::DynAnyFactory'
   object
        org.omg.CORBA.Object obj
            = orb.resolve_initial_references("DynAnyFactory");
        dyn_fact
            = org.omg.DynamicAny.DynAnyFactoryHelper.narrow(obj);

        org.omg.DynamicAny.DynAny dyn_a
            = dyn_fact.create_dyn_any(a);
        TypeCode tcode = dyn_a.type();

        if (tcode.kind()==TCKind.tk_enum)
        {
            org.omg.DynamicAny.DynEnum dyn_e
                = org.omg.DynamicAny.DynEnumHelper.narrow(dyn_a);
            String s = dyn_e.get_as_string();
            System.out.println(s);
            dyn_e.destroy();
        }
        // other cases follow
        // ...
    }

    catch (SystemException se) {
        // error: handle exception
    }
    catch (Exception ex) {
        // error: handle exception
    }
}
```

DynStruct

The DynStruct interface is used for struct and exception types. The interface is defined as follows:

```
module DynamicAny {
// ...
    typedef string FieldName;

    struct NameValuePair{
        FieldName id;
        any value;
    };
    typedef sequence<NameValuePair> NameValuePairSeq;

    struct NameDynAnyPair {
        FieldName id;
        DynAny value;
    };
    typedef sequence<NameDynAnyPair> NameDynAnyPairSeq;

    interface DynStruct : DynAny{
        FieldName current_member_name()
            raises(TypeMismatch, InvalidValue);
        CORBA::TCKind current_member_kind()
            raises(TypeMismatch, InvalidValue);
        NameValuePairSeq get_members();
        void set_members (in NameValuePairSeq value)
            raises(TypeMismatch, InvalidValue);
        NameDynAnyPairSeq get_members_as_dyn_any();
        void set_members_as_dyn_any(
            in NameDynAnyPairSeq value
        ) raises(TypeMismatch, InvalidValue);
    };
};
```

The DynStruct interface defines the following operations:

- set_members() and get_members() are used to get and set member values in a DynStruct. Members are defined as a NameValuePairSeq sequence of name-value pairs, where each name-value pair consists of the member's name as a string, and an any that contains its value.

- `current_member_name()` returns the name of the member at the current position, as established by `DynAny` base interface operations. Because member names are optional in type codes, `current_member_name()` might return an empty string.
- `current_member_kind()` returns the `TCKind` value of the current `DynStruct` member's type code.
- `get_members_as_dyn_any()` and `set_members_as_dyn_any()` are functionally equivalent to `get_members()` and `set_members()`, respectively. They operate on sequences of name-`DynAny` pairs. Use these operations if you work extensively with `DynStruct` objects; doing so allows you to avoid converting a constructed `DynAny` into an `any` before using the operations to get or set struct members.

The following code iterates over members in a `DynStruct` and passes each member over to `eval_member()` for further decomposition:

Example 33: *Using a DynStruct*

```
import org.omg.CORBA.*;
import org.omg.CORBA.ORBPackage.*;
//...
org.omg.DynamicAny.DynStruct dyn_s = ...;
TypeCode tcode = dyn_s.type();
CORBA::ULong counter = tcode.member_count();

for (CORBA::ULong i = 0; i < counter; i++) {
    org.omg.DynamicAny.DynAny member = dyn_s.current_component();
    eval_member(member);
    dyn_s.next();
}
```

DynUnion

The DynUnion interface enables access to any values of union type:

```
module DynamicAny {
    //...
    typedef string FieldName;

    interface DynUnion : DynAny {
        DynAny get_discriminator();
        void set_discriminator(in DynAny d) raises(TypeMismatch);
        void set_to_default_member() raises(TypeMismatch);
        void set_to_no_active_member() raises(TypeMismatch);
        boolean has_no_active_member() raises(InvalidValue);
        CORBA::TCKind discriminator_kind();
        DynAny member() raises(InvalidValue);
        FieldName member_name() raises(InvalidValue);
        CORBA::TCKind member_kind() raises(InvalidValue);
    };
};
```

The DynUnion interface defines the following operations:

get_discriminator() returns the current discriminator value of the DynUnion.

set_discriminator() sets the discriminator of the DynUnion to the specified value. If the type code of the parameter is not equivalent to the type code of the union's discriminator, the operation raises TypeMismatch.

set_to_default_member() sets the discriminator to a value that is consistent with the value of the default case of a union; it sets the current position to zero and causes component_count to return 2. Calling set_to_default_member() on a union that does not have an explicit default case raises TypeMismatch.

set_to_no_active_member() sets the discriminator to a value that does not correspond to any of the union's case labels; it sets the current position to zero and causes component_count to return 1. Calling set_to_no_active_member() on a union that has an explicit default case or on a union that uses the entire range of discriminator values for explicit case labels raises TypeMismatch.

has_no_active_member() returns true if the union has no active member (that is, the union's value consists solely of its discriminator, because the discriminator has a value that is not listed as an explicit case label). Calling this operation on a union that has a default case returns false. Calling this operation on a union that uses the entire range of discriminator values for explicit case labels returns false.

discriminator_kind() returns the `TCKind` value of the discriminator's `TypeCode`.

member() returns the currently active member. If the union has no active member, the operation raises `InvalidValue`. Note that the returned reference remains valid only as long as the currently active member does not change. Using the returned reference beyond the life time of the currently active member raises `OBJECT_NOT_EXIST`.

member_name() returns the name of the currently active member. If the union's type code does not contain a member name for the currently active member, the operation returns an empty string. Calling `member_name()` on a union that does not have an active member raises `InvalidValue`.

member_kind() returns the `TCKind` value of the currently active member's `TypeCode`. Calling this operation on a union that does not have a currently active member raises `InvalidValue`.

DynSequence and DynArray

The interfaces for `DynSequence` and `DynArray` are virtually identical:

```
module DynamicAny {
    //...
    typedef sequence<any> AnySeq;
    typedef sequence<DynAny> DynAnySeq;

    interface DynArray : DynAny {
        AnySeq get_elements();
        void set_elements(in AnySeq value)
            raises (TypeMismatch, InvalidValue);
        DynAnySeq get_elements_as_dyn_any();
        void set_elements_as_dyn_any(in DynAnySeq value)
            raises (TypeMismatch, InvalidValue);
    };
```

```
    interface DynSequence : DynAny {
        unsigned long get_length();
        void set_length(in unsigned long len)
            raises(InvalidValue);

        // remaining operations same as for DynArray
        // ...
    };
};
```

You can get and set element values in a `DynSequence` or `DynArray` with operations `get_elements()` and `set_elements()`, respectively. Members are defined as an `AnySeq` sequence of `any` objects.

Operations `get_elements_as_dyn_any()` and `set_elements_as_dyn_any()` are functionally equivalent to `get_elements()` and `set_elements()`; unlike their counterparts, they return and accept sequences of `DynAny` elements.

`DynSequence` has two of its own operations:

get_length() returns the number of elements in the sequence.

set_length() sets the number of elements in the sequence.

If you increase the length of a sequence, new elements are appended to the sequence and default-initialized. If the sequence's current position is undefined (equal to -1), increasing the sequence length sets the current position to the first of the new elements. Otherwise, the current position is not affected.

If you decrease the length of a sequence, `set_length()` removes the elements from its end.

You can access elements with the iteration operations described in "Iterating Over DynAny Components" on page 356. For example, the following code iterates over elements in a `DynArray`:

```
//Java
import org.omg.CORBA.*;
import org.omg.CORBA.ORBPackage.*;
//...
org.omg.DynamicAny.DynArray dyn_array = ...;
TypeCode tcode = dyn_array.type();
CORBA::ULong counter = tcode.length();

for (CORBA::ULong i = 0; i < counter; i++) {
    org.omg.DynamicAny.DynAny elem
        = dyn_array.current_component();
    eval_member(member);
    dyn_array.next();
}
```

DynFixed

The `DynFixed` interface lets you manipulate an `any` that contains fixed-point values.

```
interface DynAny{
...
    interface DynFixed : DynAny{
        string get_value();
        void set_value(in string val)
            raises (TypeMismatch, InvalidValue);
    };
};
```

The `DynFixed` interface defines the following operations:

get_value() returns the value of a `DynFixed` as a string.

set_value() sets the value of a `DynFixed`. If `val` is an uninitialized string or contains a fixed point literal that exceeds the scale of `DynFixed`, the `InvalidValue` exception is raised. If `val` is not a valid fixed point literal, the `TypeMismatch` exception is raised.

DynValue

The DynValue interface lets you manipulate an any that contains a value type (excluding boxed value types):

```
module DynamicAny {
    //...
    typedef string FieldName;

    struct NameValuePair
    {
        FieldName id;
        any       value;
    };
    typedef sequence<NameValuePair> NameValuePairSeq;

    struct NameDynAnyPair
    {
        FieldName id;
        DynAny    value;
    };
    typedef sequence<NameDynAnyPair> NameDynAnyPairSeq;

    interface DynValue : DynAny
    {
        FieldName current_member_name()
            raises (TypeMismatch, InvalidValue);
        CORBA::TCKind current_member_kind()
            raises (TypeMismatch, InvalidValue);
        NameValuePairSeq get_members();
        void set_members(in NameValuePairSeq values)
            raises (TypeMismatch, InvalidValue);
        NameDynAnyPairSeq get_members_as_dyn_any();
        void set_members_as_dyn_any(in NameDynAnyPairSeq value)
            raises (TypeMismatch, InvalidValue);
    };
};
```

The DynValue interface defines the following operations:

current_member_name() returns the name of the value type member indexed by the current position.

current_member_kind() returns the type code kind for the value type member indexed by the current position.

get_members() returns the complete list of value type members in the form of a `NameValuePairSeq`.

set_members() sets the contents of the value type members using a `NameValuePairSeq`.

get_members_as_dyn_any() is similar to `get_members()`, except that the result is returned in the form of a `NameDynAnyPairSeq`.

set_members_as_dyn_any() is similar to `set_members()`, except that the contents are set using a `NameDynAnyPairSeq`.

DynValueBox

The `DynValueBox` interface lets you manipulate an `any` that contains a boxed value type:

```
module DynamicAny {
    //...
    interface DynValueBox : DynAny
    {
        any get_boxed_value();
        void set_boxed_value(in any val)
            raises (TypeMismatch);
        DynAny get_boxed_value_as_dyn_any();
        void set_boxed_value_as_dyn_any(in DynAny val)
            raises (TypeMismatch);
    };
};
```

The `DynValue` interface defines the following operations:

get_boxed_value() returns the boxed value as an `any`.

set_boxed_value() sets the boxed value as an `any`.

get_boxed_value_as_dyn_any() returns the boxed value as a `DynAny`.

set_boxed_value_as_dyn_any() sets the boxed value as a `DynAny`.

CHAPTER 15

Generating Interfaces at Runtime

The dynamic invocation interface lets a client invoke on objects whose interfaces are known only at runtime; similarly, the dynamic skeleton interface lets a server process requests on objects whose interfaces are known only at runtime.

An application's IDL usually describes interfaces to all the CORBA objects that it requires at runtime. Accordingly, the IDL compiler generates the stub and skeleton code that clients and servers need in order to issue and process requests. The client can issue requests only on those objects whose interfaces are known when the client program is compiled; similarly, the server can process requests only on those objects that are known when the server program is compiled.

Some applications cannot know ahead of time which objects might be required at runtime. In this case, Orbix provides two interfaces that let you construct stub and skeleton code at runtime, so clients and servers can issue and process requests on those objects:

- The *dynamic invocation interface* (DII) builds stub code for a client so it can call operations on IDL interfaces that were unknown at compile time.

- The *dynamic skeleton interface* (DSI) builds skeleton code for a server, so it can receive operation or attribute invocations on an object whose IDL interface is unknown at compile time.

In this chapter

This chapter discusses the following topics:

Using the DII

Overview

Some application programs and tools must be able to invoke on objects whose interfaces cannot be determined ahead of time—for example, browsers, gateways, management support tools, and distributed debuggers.

With DII, invocations can be constructed at runtime by specifying the target object reference, the operation or attribute name, and the parameters to pass. A server that receives a dynamically constructed invocation request does not differentiate between it and static requests.

Clients that use DII

Two types of client programs commonly use the DII:

- A client interacts with the interface repository to determine a target object's interface, including the name and parameters of one or all of its operations, then uses this information to construct DII requests.
- A client, such as a gateway, receives the details of a request. In the case of a gateway, the request details might arrive as part of a network package. The gateway can then translate this into a DII call without checking the details with the interface repository. If a mismatch occurs, an exception is raised to the gateway, which in turn can report an error to the caller.

Steps

To invoke on an object with DII, follow these steps:

1. Construct a `Request` object with the operation's signature.
2. Invoke the request.
3. Retrieve results of the operation.

Example IDL

The bank example is modified here to show how to use the DII. The `Bank::newAccount()` operation now takes an `inout` parameter that sets a new account's initial balance:

```
// IDL
interface Account {
    readonly attribute float balance;

    void makeDeposit(in float f);
    void makeWithdrawal(in float f);
};

interface Bank {
    exception Reject {string reason;};

    // Create an account
    Account newAccount(
        in string owner,
        inout float initialBalance,
        out long status)
        raises (Reject);

    // Delete an account
    void deleteAccount(in Account a);
};
```

The following section shows how to construct a `Request` object that can deliver client requests for `newAccount()` operations such as this one:

```
bank.newAccount(ownerName, initialBalance, status);
```

In this section

This section discusses the following topics:

Constructing a Request Object

Overview

To construct a `Request` object and set its data, you must first obtain a reference to the target object. You then create a request object by invoking one of these methods on the object reference:

- _request() returns an empty request object whose signature—return type and parameters—must be set.
- _create_request() returns with a request object that can contain all the data required to invoke the desired request.

In this section

This section discusses the following topics:

_request()

Overview

You can use `_request()` to create a `Request` object in these steps:

1. Create a request object and set the name of its operation.
2. Set the operation's return type.
3. Set operation parameters and supply the corresponding arguments.
4. Set exception type codes.
5. Set the operation's context clause, if necessary.

Create a request object

Call `_request()` on the target object and specify the name of the operation to invoke:

```
// Get object reference
org.omg.CORBA.Object target = ... ;

// Create Request object for operation newAccount()
org.omg.CORBA.Request newAcctRequest =
        target._request("newAccount");
```

Set the operation's return type

After you create a `Request` object, set the `TypeCode` of the operation's return value by calling `set_return_type()` on the `Request` object. `set_return_type()` takes a single argument, the `TypeCode` constant of the return type. For example, given the `Request` object `newAcctRequest`, set the return type of its `newAccount()` operation to `Account` as follows:

```
newAcctRequest.set_return_type(_tc_Account);
```

For information about supported `TypeCodes`, see Chapter 13 on page 311.

Set operation parameters

A request object uses an `NVList` to store the data for an operation's parameters. To set the parameters in the `NVList` you need to know the operations parameters and insert the proper values in the exact order the parameters are specified in the operation's IDL. The `_request()` operation creates an empty `NVList` into which you insert the values needed by the operation.

To fill in the NVList you can use the following operations on the Request object:

```
add_in_arg();
any add_named_in_arg();
any add_inout_arg();
any add_named_inout_arg();
any add_out_arg();
any add_named_out_arg();
```

These operations return a reference to an Any. For more information on inserting values into an Any see "Using the Any Data Type" on page 325.

Example 34 on page 375 sets the parameter list for the newAccount operation.

Example 34: *Setting the parameter list*

```
//Java
newAcctRequest.add_in_arg().insert_string("Norman Fellows");
float initBal = 1000.00;
newAcctRequest.add_inout_arg().insert_float(initBal);
int status;
newAcctRequest.add_out_arg().insert_long(status);
```

The values for the out parameters of an operation do not need to be set because they will be changed when the operation returns. However, the values for all in and inout parameters must be specified.

You can also fill the NVList object using NVList::add_value(). This operation has the following signature:

```
NamedValue NVList::add_value(String item_name, Any val, int
   flags);
```

The flags parameter is set to one of the following values:

- CORBA::ARG_IN
- CORBA::ARG_INOUT
- CORBA::ARG_OUT

Set exception type codes

You must set the type codes for any exceptions defined for the *Request* object's operation. To do this use the add() operation defined for the Request object's exceptions() list.

`add()` takes the exceptions type codes as its only argument. To add the `Reject` exception to `newAcctRequest` use the following operation:

```
newAcctRequest.exceptions().add(BankPackage.RejectHelper.type())
    ;
```

If the type code for the exception was not available in the stub code, you would need to dynamically generate the exceptions type code.

Set the operation's context clause

If the IDL operation has a `context` clause, you can add a `Context` object to its `Request` object with `CORBA::Request::ctx()`.

_create_request()

Overview

You can also create a Request object by calling _create_request() on an object reference and passing the request details as arguments. The advantage of using _create_request() is that you can create a Request object that contains all of the information needed to invoke a request. _create_request() has the following signature:

```
Request _create_request(Context ctx,
                        String operation,
                        NVList arg_list,
                        NamedValue result,
                        ExceptionList exclist,
                        ContextList ctxlist);
```

At a minimum, you must provide two arguments when using _create_request():

- The name of the operation
- A NamedValue that holds the operation's return value

You can also supply a populated parameter list and a populated exception list to _create_request(). If you supply null for either list, _create_request() creates an empty list for the returned Request object. In this case you must populate the list as described above in "_request()" on page 374.

Creating the parameter list

There are two operations provided by CORBA::ORB to create the NVList passed to _create_object() to specify the Request object's parameter list:

- create_list()
- create_operation_list()

create_list()

create_list() has the fololwing signiture:

```
NVList create_list(int count);
```

The operation allocates the space for an NVList of the specified number of elements and returns a pointer to the empty NVList. You then add the required parameters using the following operation on the NVList:

```
add()
add_item()
add_value()
```

create_operation_list()

create_operation_list() extends the functionality of create_list() by creating a prefilled parameter list based on informaiton stored in the interface repository. It has the following signature:

```
NVList create_operation_list(OperationDef operation);
```

Using the OperationDef object passed as a parameter, create_operation_list() retrieves the parameter list for the specified operation from the interface repository. When create_operation_list() returns, the NVList contains one NamedValue object for each operation parameter. Each NamedValue object contains the parameter's passing mode, name, and initial value of type Any.

Once you have the prefilled parameter list, you can modify the parameters by iterating over the NVList elements with org.omg.NVList.item(). Use the appropriate insert operation to set each NamedValue's value member.

Example

The code in Example 35 constructs a parameter list using create_operation_list(). It then uses the parameter list to construct a Request object for invoking operation newAccount():

Example 35: *Create a Request object using _create_request()*

```
// get an object reference
org.omg.CORBA.Object target = ... ;

org.omg.CORBA.Request newAcctRequest;

// construct Any for return value
org.omg.CORBA.Any result_any = orb.create_any();
org.omg.CORBA.NamedValue result =
    orb.create_named_value("result", result_any, ARG_OUT.value);

// Get OperationDef object from IFR
// reference to the IFR, ifr, obtained previously
org.omg.CORBA.Contained cont = ifr.lookup("Bank::newAccount");
org.omg.CORBA.OperationDef opDef =
   org.omg.CORBA.OperationDefHelper._narrow(cont.in());

// Initialize the parameter list
org.omg.CORBA.NVList paramList =
   orb.create_operation_list(opDef, paramList);
paramList.item(0).value.insert_string("Norman Fellows");
float initBal = 1000.00;
paramList.item(1).value.insert_float(initBal);
int status;
paramList.item(2).value.insert_long(status);

// Construct the Request object
newAcctRequest = target._create_request(null, "newAccount",
   paramList, result);
```

Invoking a Request

After you set a `Request` object's data, you can use one of severalmethods to invoke the request on the target object. The following methods are invoked on a `Request` object:

invoke() blocks the client until the operation returns with a reply. Exceptions are handled the same as static function invocations.

send_deferred() sends the request to the target object and allows the client to continue processing while it awaits a reply. The client must poll for the request's reply (see "Invoking Deferred Synchronous Requests" on page 382).

send_oneway() invokes one-way operations. Because no reply is expected, the client resumes processing immediately after the invocation.

The following methods are invoked on the ORB, and take a sequence of requests:

send_multiple_requests_deferred() calls multiple deferred synchronous operations.

send_multiple_requests_oneway() calls multiple oneway operations simultaneously.

For example:

Example 36: *Invoking on a request*

```
try {
    request.invoke())
}
catch (org.omg.CORBA.SystemException se) {
    System.out.println( "Unexpected exception"  + se );
}
```

Retrieving Request Results

When a request returns, Orbix updates `out` and `inout` parameters in the `Request` object's `NVList`. To get an operation's output values:

1. Call `arguments()` on the `Request` object to get a reference to its `NVList`.
2. Iterate over the `NamedValue` items in the `Request` object's `NVList` by successively calling `item()` on the `NVList`. Each call to this methods returns a `NamedValue` reference.
3. Call `value()` on the `NamedValue` to get a pointer to the `Any` value for each parameter.
4. Extract the parameter values from the `Any`.

To get an operation's return value, call `return_value()` on the request object. This operation returns the request's return value as an `any`.

For example, the following code gets an object reference to the new account returned by the `newAccount()` operation:

Example 37: *Obtaining the return value from a request object*

```
org.omg.CORBA.Object newAccount;
org.omg.CORBA.Any acct = request.return_value();
newAccount = acct.extract_Object();
```

Invoking Deferred Synchronous Requests

You can use the DII to make *deferred synchronous* operation calls. A client can call an operation, continue processing in parallel with the operation, then retrieve the operation results when required.

You can invoke a request as a deferred synchronous operation as follows:

1. Construct a `Request` object and call `send_deferred()` on it.
2. Continue processing in parallel with the operation.
3. Check whether the operation has returned by calling `poll_response()` on the `Request` object. This methods returns a non-zero value if a response has been received.
4. To get the result of the operation, call `get_response()` on the `Request` object.

Using the DSI

Overview

A server uses the dynamic skeleton interface (DSI) to receive operations or attribute invocations on an object whose IDL interface is unknown to it at compile time. With DSI, a server can build the skeleton code that it needs to accept these invocations.

The server defines a function that determines the identity of the requested object; the name of the operation and the types and values of each argument are provided by the user. The function carries out the task that is being requested by the client, and constructs and returns the result. Clients are unaware that a server is implemented with the DSI.

In this section

This section discusses the following topics:

DSI Applications

Overview

The DSI is designed to help write gateways that accept operation or attribute invocations on any specified set of interfaces and pass them to another system. A gateway can be written to interface between CORBA and some non-CORBA system. This gateway is the only part of the CORBA system that must know the non-CORBA system's protocol; the rest of the CORBA system simply issues IDL calls as usual.

Invoking on a gateway

The IIOP protocol lets an object invoke on objects in another ORB. If a non-CORBA system does not support IIOP, you can use DSI to provide a gateway between the CORBA and non-CORBA systems. To the CORBA system, this gateway appears as a CORBA-compliant server that contains CORBA objects. In reality, the server uses DSI to trap incoming invocations and translate them into calls that the non-CORBA system can understand.

Bidirectional gateways

You can use DSI and DII together to construct a bidirectional gateway. This gateway receives messages from the non-CORBA system and uses the DII to make CORBA client calls. It uses DSI to receive requests from clients on a CORBA system and translate these into messages in the non-CORBA system.

DSI has other uses. For example, a server might contain many non-CORBA objects that it wants to make available to its clients. In an application that uses DSI, clients invoke on only one CORBA object for each non-CORBA object. The server indicates that it uses DSI to accept invocations on the IDL interface. When it receives an invocation, it identifies the target object, the operation or attribute to call, and its parameters. It then makes the call on the non-CORBA object. When it receives the result, it returns it to the client.

Programming a Server to Use DSI

Overview

The DSI is implemented by servants that instantiate dynamic skeleton classes. All dynamic skeleton classes are derived from `CORBA.DynamicImplementation`:

```
package org.omg.PortableServer;
abstract public class DynamicImplementation extends Servant
{
abstract public void invoke(org.omg.CORBA.ServerRequest
   request);
}
```

Note: The ORB user must also provide an implementation to the `_all_interfaces()` method declared by the `Servant` class.

A server program uses DSI as follows:

1. Instantiates one or more DSI servants and obtains object references to them, which it makes available to clients.
2. Associates each DSI servant with a POA—for example, through a servant manager, or by registering it as the default servant.

Dynamic implementation routine

When a client invokes on a DSI-generated object reference, the POA delivers the client request as an argument to the DSI servant's `invoke()` method—also known as the *dynamic implementation routine* (DIR). `invoke()` takes a single argument, a `CORBA::ServerRequest` pseudo-object, which

encapsulates all data that pertains to the client request—the operation's signature and arguments. `CORBA::ServerRequest` maps to the following Java class:

```
package org.omg.CORBA;
public abstract class ServerRequest {

    public String operation() {
        ...
    }
    public void arguments(NVList args) {
        ...
    }
    public void set_result(Any any) {
        ...
    }
    public void set_exception(Any any) {
        ...
    }
    public abstract Context ctx();
}
```

invoke() processing

`invoke()` processing varies across different implementations, but it always includes the following steps:

1. Obtains the operation's name by calling `operation()` on the `ServerRequest` object.
2. Builds an `NVList` that contains definitions for the operation's parameters—often, from an interface definition obtained from the interface repository. Then, `invoke()` populates the `NVList` with the operation's input arguments by calling `arguments()` on the `ServerRequest` object.
3. Reconstructs the client invocation and processes it.
4. If required, sets the operation's output in one of two ways:
   - If the operation's signature defines output parameters, `invoke()` sets the `NVList` as needed. If the operation's signature defines a return value, `invoke()` calls `set_result()` on the `ServerRequest` object.

- If the operation's signature defines an exception, `invoke()` calls `set_exception()` on the `ServerRequest` object.

Note: `invoke()` can either set the operation's output by initializing its output parameters and setting its return value, or by setting an exception; however, it cannot do both.

CHAPTER 16

Using the Interface Repository

An Orbix application uses the interface repository for persistent storage of IDL interfaces and types. The runtime ORB and Orbix applications query this repository at runtime to obtain IDL definitions.

The interface repository maintains full information about the IDL definitions that have been passed to it. The interface repository provides a set of IDL interfaces to browse and list its contents, and to determine the type information for a given object. For example, given an object reference, you can use the interface repository to obtain all aspects of the object's interface: its enclosing module, interface name, attribute and operation definitions, and so on.

Benefits

These capabilities are important for a number of tools:

- Browsers that allow designers and code writers to determine what types have been defined in the system, and to list the details of chosen types.
- CASE tools that aid software design, writing, and debugging.
- Application level code that uses the dynamic invocation interface (DII) to invoke on objects whose types were not known to it at compile time. This code might need to determine the details of the object being invoked in order to construct the request using the DII.

- A gateway that requires runtime information about the type of an object being invoked.

In order to populate the interface repository with IDL definitions, run the IDL compiler with the `-R` option. For example, the following command populates the interface repository with the IDL definitions in `bank.idl`:

```
idl -R bank.idl
```

In this chapter

This chapter contains the following sections

Interface Repository Data

Interface repository data can be viewed as a set of CORBA objects, where the repository stores one object for each IDL type definition. All interface repository objects are derived from the abstract base interface `IRObject`., which is defined as follows:

```
// In module CORBA
enum DefinitionKind
{
    dk_none, dk_all,
    dk_Attribute, dk_Constant, dk_Exception, dk_Interface,
    dk_Module, dk_Operation, dk_Typedef,
    dk_Alias, dk_Struct, dk_Union, dk_Enum,
    dk_Primitive, dk_String, dk_Sequence, dk_Array,
    dk_Repository, dk_Wstring, dk_Fixed,
    dk_Value, dk_ValueBox, dk_ValueMember, dk_Native
};

...
interface IRObject
{
    // read interface
    readonly attribute DefinitionKind def_kind;

    // write interface
    void
    destroy();
};
```

Attribute `def_kind` identifies a repository object's type. For example, the `def_kind` attribute of an `interfaceDef` object is `dk_interface`. The enumerate constants `dk_none` and `dk_all` are used to search for objects in a repository. All other enumerate constants identify one of the repository object types in Table 16, and correspond to an IDL type or group of types.

`destroy()` deletes an interface repository object and any objects contained within it. You cannot call `destroy()` on the interface repository object itself or any `PrimitiveDef` object.

Abstract Base Interfaces

Besides `IRObject`, the interface repository defines four other abstract base interfaces, all of which inherit directly or indirectly from `IRObject`:

Container: The interface for container objects. This interface is inherited by all interface objects that can contain other objects, such as `Repository`, `ModuleDef` and `InterfaceDef`. These interfaces inherit from `Container`. See "Container Interface" on page 405.

Contained: The interface for contained objects. This interface is inherited by all objects that can be contained by other objects—for example, attribute definition (`AttributeDef`) objects within operation definition (`OperationDef`) objects. See "Contained Interface" on page 403.

IDLType: All interface repository interfaces that hold the definition of a type inherit directly or indirectly from this interface. See "IDL-type objects" on page 396.

TypedefDef: The base interface for the following interface repository types that have names: `StructDef`, `UnionDef`, `EnumDef`, and `AliasDef`, which represents IDL `typedef` definitions.

Repository Object Types

Objects in the interface repository support one of the IDL types in Table 16:

Table 16: *Interface Repository OIbject Types*

Object type	Description
`Repository`	The repository itself, in which all other objects are nested. A repository definition can contain definitions of other types such as module and interface. Table 17 lists all possible container components.
`ModuleDef`	A module definition is logical grouping of interfaces and value types. The definition has a name and can contain definitions of all types except `Repository`. Table 17 on page 401 lists all possible container components.
`InterfaceDef`	An interface definition has a name, a possible inheritance declaration, and can contain definitions of other types such as attribute, operation, and exception. Table 17 lists all possible container components.
`ValueDef`	A value type definition has a name, a possible inheritance declaration, and can contain definitions of other types such as attribute, operation, and exception. Table 17 lists all possible container components.
`ValueBoxDef`	A value box definition defines a value box type.
`ValueMemberDef`	A value member definition defines a member of a value.
`AttributeDef`	An attribute definition has a name, a type, and a mode to indicate whether it is readonly.

Table 16: *Interface Repository OIbject Types*

Object type	Description
`OperationDef`	An operation definition has a name, return value, set of parameters and, optionally, `raises` and `context` clauses.
`ConstantDef`	A constant definition has a name, type, and value.
`ExceptionDef`	An exception definition has a name and a set of member definitions.
`StructDef`	A struct definition has a name, and holds the definition of each of its members.
`UnionDef`	A union definition has a name, and holds a discriminator type and the definition of each of its members.
`EmumDef`	An enum definition has a name and a list of member identifiers.
`AliasDef`	An aliased definition defines a typedef definition, which has a name and a type that it maps to.
`PrimitiveDef`	A primitive definition defines primitive IDL types such as `short` and `long`, which are predefined in the interface repository.
`StringDef`	A string definition records its bound. Objects of this type are unnamed. If they are defined with a `typedef` statement, they are associated with an `AliasDef` object. Objects of this type correspond to bounded strings.
`SequenceDef`	Each sequence type definition records its element type and its bound, where a value of zero indicates an unbounded sequence type. Objects of this type are unnamed. If they are defined with a `typedef` statement, they have an associated `AliasDef` object.

Table 16: *Interface Repository OIbject Types*

Object type	Description
`ArrayDef`	Each array definition records its length and its element type. Objects of this type are unnamed. If they are defined with a `typedef` statement, they are associated with an `AliasDef` object. Each `ArrayDef` object represents one dimension; multiple `ArrayDef` objects can represent a multi-dimensional array type.

Given an object of any interface repository type, you can obtain its full interface definition. For example, `InterfaceDef` defines operations or attributes to determine an interface's name, its inheritance hierarchy, and the description of each operation and each attribute.

Figure 20 shows the hierarchy for all interface repository objects.

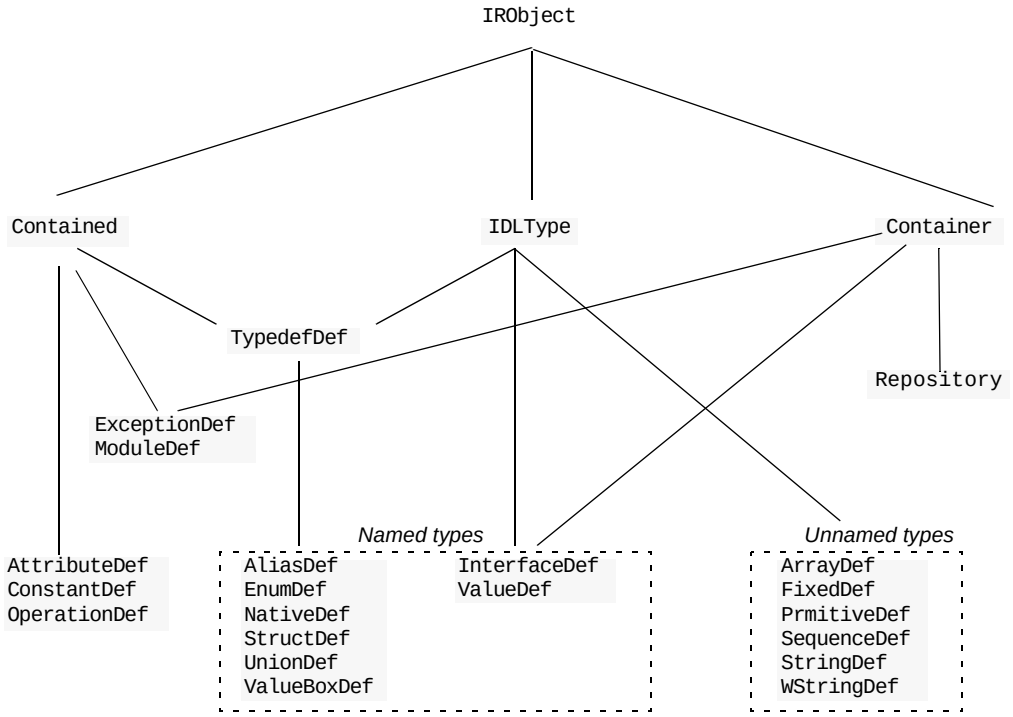


Figure 20: *Hierarchy of interface repository objects*

IDL-type objects

Most repository objects represent IDL types—for example, `InterfaceDef` objects represent IDL interfaces, `StructDef` interfaces represent struct definitions, and so on. These objects all inherit, directly or indirectly, from the abstract base interface `IDLType`:

```
// In module CORBA
interface IDLType : IRObject {
    readonly attribute TypeCode type;
};
```

This base interface defines a single attribute that contains the `TypeCode` of the defined type.

IDL-type objects are themselves subdivided into two groups:

- Named types
- Unnamed types

Named types

The interface repository can contain these named IDL types:

```
AliasDef        StructDef
EnumDef         UnionDef
InterfaceDef    ValueBoxDef
NativeDef       ValueDef
```

For example, the following IDL defines `enum` type `UD` and `typedef` type `AccountName`, which the interface repository represents as named object types `EnumDef` and `AliasDef` objects, respectively:

```
// IDL
enum UD {UP, DOWN};
typedef string AccountName;
```

The following named object types inherit from the abstract base interface `TypedefDef`:

```
AliasDef                    StructDef
EnumDef                     ValueBoxDef
NativeDef                   UnionDef
```

`TypedefDef` is defined as follows:

```
// IDL
// In module CORBA
interface TypedefDef : Contained, IDLType {
};
```

`TypedefDef` serves the sole purpose of enabling its derived object types to inherit `Contained` and `IDLType` attributes and operations:

- Attribute `Contained::name` enables access to the object's name. For example, the IDL `enum` definition `UD` shown earlier is represented by the repository object `EnumDef`, whose inherited `name` attribute is set to `UD`.
- Operation `Contained::describe()` gets a detailed description of the object. For more information about this operation, see "Repository Object Descriptions" on page 407.

Interfaces InterfaceDef and ValueDef are also named object types that inherit from three base interfaces: Contained, Container, and IDLType.

Because IDL object and value references can be used like other types, IntefaceDef and ValueDef inherit from the base interface IDLType. For example, given the IDL definition of interface Account, the interface repository creates an InterfaceDef object whose name attribute is set to Account. This name can be reused as a type.

Unnamed types

The interface repository can contain the following unnamed object types:

ArrayDef	SequenceDef
FixedDef	StringDef
PrimitiveDef	WStringDef

Getting an object's idl type

Repository objects that inherit the IDLType interface have their own operations for identifying their type; you can also get an object's type through the TypeCode interface. Repository objects such as AttributeDef that do not inherit from IDLType have their own TypeCode or IDLType attributes that enable access to their types.

For example the following IDL interface definition defines the return type of operation getLongAddress as a string sequence:

```
// IDL
interface Mailer {
    string getLongAddress();
};
```

getLongAddress() maps to an object of type OperationDef in the repository. You can query this object for its return type's definition—string—in two ways:

Method 1:

1. Get the object's OperationDef::result_def attribute, which is an object reference of type IDLType.
2. Get the IDLType's def_kind attribute, which is inherited from IRObject. In this example, def_kind resolves to dk_primitive.
3. Narrow the IDLType to PrimtiveDef.

4. Get the `PrimtiveDef`'s `kind` attribute, which is a `PrimtiveKind` of `pk_string`.

Method 2:

1. Get the object's `OperationDef::result` attribute, which is a `TypeCode`.
2. Obtain the `TypeCode`'s `TCKind` through its `kind()` operation. In this example, the `TCKind` is `tk_string`.

Containment in the Interface Repository

Most IDL definitions contain or are contained by other definitions, and the interface repository defines its objects to reflect these relationships. For example, a module typically contains interface definitions, while interfaces themselves usually contain attributes, operations, and other definition types.

Containment interfaces

The interface repository abstracts the properties of containment into two abstract base interfaces:

- `Contained`
- `Container`

These interfaces provide operations and attributes that let you traverse the hierarchy of relationships in an interface repository in order to list its contents, or ascertain a given object's container. Most repository objects are derived from one or both of `Container` or `Contained`; the exceptions are instances of `PrimitiveDef`, `StringDef`, `SequenceDef`, and `ArrayDef`.

Example

In the following IDL, module `Finance` is defined with two interface definitions, `Bank` and `Account`. In turn, interface `Account` contains attribute and operation definitions:

```
// IDL
module Finance {
    interface Account {
        readonly attribute float balance;
        void makeDeposit(in float amount);
        void makeWithdrawal(in float amount);
    };
    interface Bank {
        Account newAccount();
    };
};
```

The corresponding interface repository objects for these definitions are each described as `Container` or `Contained` objects. Thus, the interface repository represents module `Finance` as a `ModuleDef` container for `InterfaceDef`

objects Account and Bank; these, in turn, serve as containers for their respective attributes and operations. ModuleDef object Finance is also viewed as a contained object within the container object RepositoryDef.

Containment properties of interface repository objects

Table 17 shows the relationship between Container and Contained objects in the interface repository.

Table 17: *Container and Contained Objects in the Interface Repository*

Container object type	Contained Objects
Repository	ConstantDef TypedefDef ExceptionDef InterfaceDef* ModuleDef* ValueDef*
ModuleDef	ConstantDef TypedefDef ExceptionDef ModuleDef* InterfaceDef* ValueDef*
InterfaceDef	ConstantDef TypedefDef ExceptionDef AttributeDef OperationDef
ValueDef	ConstantDef TypedefDef ExceptionDef AttributeDef OperationDef ValueMemberDef

* Also a Container object

Only a Repository is a pure Container. An interface repository server has only one Repository object, and it contains all other definitions.

Objects of type `ModuleDef`, `InterfaceDef`, and `ValueDef` are always contained within a `Repository`, while `InterfaceDef`, and `ValueDef` can also be within a `ModuleDef`; these objects usually contain other objects, so they inherit from both `Container` and `Contained`.

All other repository object types inherit only from `Contained`.

Contained Interface

The Contained interface is defined as follows:

```
//IDL
typedef string VersionSpec;

interface Contained : IRObject
{
    // read/write interface
    attribute RepositoryId id;
    attribute Identifier name;
    attribute VersionSpec version;

    // read interface
    readonly attribute Container defined_in;
    readonly attribute ScopedName absolute_name;
    readonly attribute Repository containing_repository;

    struct Description
    {
        DefinitionKind kind;
        any value;
    };

    Description
    describe();

    // write interface
    void
    move(
        in Container new_container,
        in Identifier new_name,
        in VersionSpec new_version
    );
};
```

Name attribute

Attribute Contained::name is of type Identifier, a typedef for a string, and contains the IDL object's name. For example, module Finance is represented in the repository by a ModuleDef object. Its inherited ModuleDef::name attribute resolves to the string Finance. Similarly the makeWithdrawal operation is represented by an OperationDef object whose OperationDef::name attribute resolves to makeWithdrawal.

defined_in attribute

Contained also defines the attribute defined_in, which stores a reference to an object's Container. Because IDL definitions within a repository must be unique, defined_in stores a unique Container reference. However, given inheritance among interfaces, an object can be contained in multiple interfaces. For example, the following IDL defines interface CurrentAccount to inherit from interface Account:

```
//IDL
// in module Finance
interface CurrentAccount : Account {
    readonly attribute overDraftLimit;
};
```

balance attribute

Given this definition, attribute balance is contained in interfaces Account and CurrentAccount; however, attribute balance is defined only in the base interface Account. Thus, if you invoke AttributeDef::defined_in() on either Account::balance or CurrentAccount::balance, it always returns Account as the Container object.

A Contained object can include more than containment information. For example, an OperationDef object has a list of parameters associated with it and details of the return type. The operation Contained::describe() provides access to these details by returning a generic Description structure (see "Repository Object Descriptions" on page 407).

Container Interface

Interface `Container` is defined as follows:

```
//IDL
enum DefinitionKind
{
    dk_none, dk_all,
    dk_Attribute, dk_Constant, dk_Exception, dk_Interface,
    dk_Module, dk_Operation, dk_Typedef,
    dk_Alias, dk_Struct, dk_Union, dk_Enum,
    dk_Primitive, dk_String, dk_Sequence, dk_Array,
    dk_Repository, dk_Wstring, dk_Fixed,
    dk_Value, dk_ValueBox, dk_ValueMember, dk_Native
};
...

typedef sequence<Contained> ContainedSeq;

interface Container : IRObject
{
    // read interface
    ...

    Contained
    lookup(
        in ScopedName search_name
    );

    ContainedSeq
    contents(
        in DefinitionKind limit_type,
        in boolean exclude_inherited
    );

    ContainedSeq
    lookup_name (
        in Identifier search_name,
        in long levels_to_search,
        in DefinitionKind limit_type,
        in boolean exclude_inherited
    );
```

```
    struct Description
    {
        Contained contained_object;
        DefinitionKind kind;
        any value;
    };
    typedef sequence<Description> DescriptionSeq;

    DescriptionSeq
    describe_contents(
        in DefinitionKind limit_type,
        in boolean exclude_inherited,
        in long max_returned_objs
    );

    // write interface

    ... // operations to create container objects
};
```

lookup operations

The container interface provides four lookup operations that let you browse a given container for its contents: `lookup()`, `lookup_name()`, `contents()`, and `describe_contents()`. For more information about these operations, see "Browsing and listing repository contents" on page 410.

Repository Object Descriptions

Each repository object, in addition to identifying itself as a `Contained` or `Container` object, also maintains the details of its IDL definition. For each contained object type, the repository defines a structure that stores these details. Thus, a `ModuleDef` object stores the details of its description in a `ModuleDescription` structure, an `InterfaceDef` object stores its description in an `InterfaceDescription` structure, and so on.

How to obtain object descriptions

You can generally get an object's description in two ways:

- The interface for each contained object type often defines attributes that get specific aspects of an object's description. For example, attribute `OperationDef::result` gets an operation's return type.
- You can obtain all the information stored for a given object through the inherited operation `Contained::describe()`, which returns the general purpose structure `Contained::Description`. This structure's `value` member is of type `any`, whose value stores the object type's structure.

For example, interface `OperationDef` has the following definition:

```
interface OperationDef : Contained
{
    readonly attribute TypeCode result;
    attribute IDLType result_def;
    attribute ParDescriptionSeq params;
    attribute OperationMode mode;
    attribute ContextIdSeq contexts;
    attribute ExceptionDefSeq exceptions;
};
```

Accessing attributes

Interface `OperationDef` defines a number of attributes that allow direct access to specific aspects of an operation, such as its parameters (`params`) and return type (`result_def`).

Invoking describe()

In a distributed environment, it is often desirable to obtain all information about an operation in a single step by invoking `describe()` on the `OperationDef` object. This operation returns a `Contained::Description` whose two members, `kind` and `value`, are set as follows:

kind is set to `dk_Operation`.

value is an any whose `TypeCode` is set to `_tc_OperationDescription`. The any's value is an `OperationDescription` structure, which contains all the required information about an operation:

```
// IDL
struct OperationDescription
{
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    TypeCode result;
    OperationMode mode;
    ContextIdSeq contexts;
    ParDescriptionSeq parameters;
    ExcDescriptionSeq exceptions;
};
```

OperationDescription structure

`OperationDescription` members store the following information:

`name`	The operation's name. For example, for operation `Account::makeWithdrawal()`, `name` contains `makeWithdrawal`.
`id`	`RepositoryId` for the `OperationDef` object.
`defined_in`	The `RepositoryId` for the parent `Container` of the `OperationDef` object.
`version`	Currently not supported. When implemented, this member allows the interface repository to distinguish between multiple versions of a definition with the same name.
`result`	The `TypeCode` of the result returned by the defined operation.

`mode`	Specifies whether the operation returns (`OP_NORMAL`) or is oneway (`OP_ONEWAY`).
`contexts`	Lists the context identifiers specified in the operation's context clause.
`parameters`	A sequence of `ParameterDescription` structures that contain details of each operation parameter.
`exceptions`	A sequence of `ExceptionDescription` structures that contain details of the exceptions specified in the operation's `raises` clause.

TypeDescription structure

Several repository object types use the `TypeDescription` structure to store their information: `EnumDef`, `UnionDef`, `AliasDef`, and `StructDef`.

FullInterfaceDescription and FullValueDescription structures

Interfaces `InterfaceDef` and `ValueDef` contain extra description structures, `FullInterfaceDescription` and `FullValueDescription`, respectively. These structures let you obtain a full description of the interface or value and all its contents in one step. These structures are returned by operations `InterfaceDef::describe_interface()` and `ValueDef::describe_value()`.

Retrieving Repository Information

You can retrieve information from the interface repository in three ways:

- Given an object reference, find its corresponding `InterfaceDef` object and query its details.
- Given an object reference to a `Repository`, browse its contents.
- Given a `RepositoryId`, obtain a reference to the corresponding object in the interface repository and query its details.

Getting a CORBA object's interface

Given a reference to a CORBA object, you can obtain its interface from the interface repository by invoking `_get_interface()` on it. For example, given CORBA object `objVar`, you can get a reference to its corresponding `InterfaceDef` object as follows:

```
org.omg.CORBA.InterfaceDef ifVar =
                    objVar._get_interface();
```

The member function `_get_interface()` returns a reference to an object within the interface repository. You can then use this reference to browse the repository, and to obtain the details of an interface definition.

Browsing and listing repository contents

After you obtain a reference to a `Repository` object, you can browse or list its contents. To obtain a `Repository`'s object reference, invoke `resolve_initial_references("InterfaceRepository")` on the ORB. This returns an object reference of type `CORBA::Object`, which you narrow to a `CORBA::Repository` reference.

The abstract interface `Container` has four operations that enable repository browsing:

- `lookup()`
- `lookup_name()`
- `contents()`
- `describe_contents()`

Finding repository objects

`Container` operations `lookup()` and `lookup_name()` are useful for searching the contents of a repository for one or more objects.

lookup() conducts a search for a single object based on the supplied `ScopedName` argument, which contains the entity's name relative to other repository objects. A `ScopedName` that begins with `::` is an absolute scoped name—that is, it uniquely identifies an entity within a repository—for example, `::Finance::Account::makeWithdrawal`. A `ScopedName` that does not begin with `::` identifies an entity relative to the current one.

For example, if module `Finance` contains attribute `Account::balance`, you can get a reference to the operation's corresponding `AttributeDef` object by invoking the module's `lookup()` operation:

```
org.omg.CORBA.Contained cVar;
cVar = moduleVar.lookup("Account::balance");
```

The `ScopedName` argument that you supply can specify to search outside the cope of the actual container on which you invoke `lookup()`. For example, the following statement invokes `lookup()` on an `InterfaceDef` in order to start searching for the `newAccount` operation from the `Repository` container:

```
org.omg.CORBA.Contained_var cVar;
cVar = ifVar.lookup("::Finance::Bank::newAccount");
```

lookup_name() searches the target container for objects that match a simple unscoped name. Because the name might yield multiple matches, `lookup()` returns a sequence of `Contained` objects. `lookup_name()` takes the following arguments:

`search_name`	A string that specifies the name of the objects to find. You can use asterisks (`*`) to construct wildcard searches.
`levels_to_search`	Specifies the number of levels of nested containers to include in the search. `1` restricts searching to the current object. `-1` specifies an unrestricted search.
`limit_type`	Supply a `DefinitionKind` enumerator to include a specific type of repository object in the returned sequence. For example, set `limit_type` to `dk_operation` to find only operations. To return all objects, supply `dk_all`. You can also supply `dk_none` to match no repository objects, and `dk_Typedef`, which encompasses `dk_Alias`, `dk_Struct`, `dk_Union`, and `dk_Enum`.

`exclude_inherited` Valid only for `InterfaceDef` and `ValueDef` objects. Supply `TRUE` to exclude inherited definitions, `FALSE` to include.

Unlike `lookup()`, `lookup_name()` searches are confined to the target container.

Getting object descriptions

`Container` operations `contents()` and `describe_contents()` let you obtain object descriptions:

contents() returns a sequence of `Contained` objects that belong to the `Container`. You can use this operation to search a given container for a specific object. When it is found, you can call `Contained::describe()`, which returns a `Contained::Description` for the contained object (see "Repository Object Descriptions" on page 407).

describe_contents() combines operations `Container::contents()` and `Contained::describe()`, and returns a sequence of `Contained::Description` structures, one for each of the `Contained` objects found.

You can limit the scope of the search by `contents()` and `describe_contents()` by setting one or more of the following arguments:

`limit_type` Supply a `DefinitionKind` enumerator to limit the contents list to a specific type of repository object. To return all objects, supply `dk_all`. You can also supply `dk_none` to match no repository objects, and `dk_Typedef`, which encompasses `dk_Alias`, `dk_Struct`, `dk_Union`, and `dk_Enum`.

`exclude_inherited` Valid only for `InterfaceDef` and `ValueDef` objects. Supply `TRUE` to exclude inherited definitions from the contents listing, `FALSE` to include.

`max_returned_objs` Available only for `describe_contents()`, this argument specifies the maximum length of the sequence returned.

Finding an object using its repository id

You can use a repository ID to find any object in a repository by invoking `Container::lookup_id()` on that repository. `lookup_id()` returns a reference to a `Contained` object, which can be narrowed to the appropriate object reference type.

Sample Usage

This section contains code that uses the interface repository; it prints the list of operation names and attribute names that are defined in a given object's interface.

```
import org.omg.CORBA.*;
import java.io.*;
import org.omg.CORBA.InterfaceDefPackage.*;

int i;
Repository repository;
Contained contained;
InterfaceDef interface;
FullInterfaceDescription full;
org.omg.CORBA.Object obj;

try {
    // get an object reference to the IFR
    obj = orb.resolve_initial_references("InterfaceRepository");
    repository = RepositoryHelper.narrow(obj);

    // get the interface definition
    contained = repository.lookup("grid");
    interface = InterfaceDefHelper.narrow(contained);

    // get a full interface description
    full = interface.describe_interface();

    // print out operation names
    System.out.println ("The operation names are:");
    for (i=0; i < full.operations.length(); i++)
        System.out.println (full.operations[i].name);

    // print out the attribute names
    System.out.println ("The attribute names are:");

    for (i=0; i < full.attributes.length(); i++)
        System.out.println (full.attributes[i].name);
}

catch (SystemException ex) {
    ...
}
```

The example can be extended by finding the `OperationDef` object for an operation called `doit()`. Operation `Container::lookup_name()` can be used as follows:

```
Contained[] opSeq;
OperationDef doitOp;

try {
    System.out.println ("Looking up operation doit()");
    opSeq = interface.lookup_name(
                    "doit", 1, dk_Operation, 0);

    if (opSeq.length() != 1) {
        System.out.println ("Incorrect result for
   lookup_name()");
        exit(1);
    } else {
        // Narrow the result to an OperationDef
        doitOp =
            OperationDefHelper.narrow(opSeq[0])
    }
    ...
}
catch (SystemException ex) {
    ...
}
```

Repository IDs and Formats

Each interface repository object that describes an IDL definition has a repository ID. A repository ID globally identifies an IDL module, interface, constant, typedef, exception, attribute, or operation definition. A repository ID is simply a string that identifies the IDL definition.

Three formats for repository IDs are defined by CORBA. However, repository IDs are not, in general, required to be in one of these formats:

- OMG IDL
- DCE UUID
- LOCAL

OMG IDL

The default format used by Orbix, the OMG IDL format is derived from the IDL definition's scoped name:

`IDL:`*`identifier[/identifier]...:version-number`*

This format contains three colon-delimited components:

- The first component identifies the repository ID format as the OMG IDL format.
- A list of identifiers specifies the scoped name, substituting backslash (`/`) for double colon (`::`).
- *`version-number`* contains a version number with the following format:

  `major.minor`

For example, given the following IDL definitions:

```
// IDL
interface Account {
    readonly attribute float balance;
    void makeDeposit(in float amount);
};
```

The IDL format repository ID for attribute `Account::balance` looks like this:

`IDL:Account/balance:1.0`

DCE UUID

The DCE UUID has the following format:

`DCE:`*`UUID`*`:`*`minor-version-number`*

LOCAL

Local format IDs are for local use within an interface repository and are not intended to be known outside that repository. They have the following format:

LOCAL:*ID*

Local format repository IDs can be useful in a development environment as a way to avoid conflicts with repository IDs that use other formats.

Controlling Repository IDs with Pragma Directives

You can control repository ID formats with pragma directives in an IDL source file. Specifically, you can use pragmas to set the repository ID for a specific IDL definition, and to set prefixes and version numbers on repository IDs.

You can insert prefix and version pragma statements at any IDL scope; the IDL compiler assigns the prefix or version only to objects that are defined within that scope. Prefixes and version numbers are not applied to definitions in files that are included at that scope. Typically, prefixes and version numbers are set at global scope, and are applied to all repository IDs.

ID pragma

You can explicitly associate an interface repository ID with an IDL definition, such as an interface name or typedef. The definition can be fully or partially scoped and must conform with one of the IDL formats approved by the OMG (see "Repository IDs and Formats" on page 416).

For example, the following IDL assigns repository ID `idl:test:1.1` to interface `test`:

```
module Y {
    interface test {
        // ...
    };
    #pragma ID test "idl:test:1.1"
};
```

Prefix pragma

The IDL `prefix` pragma lets you prepend a unique identifier to repository IDs. This is especially useful in ensuring against the chance of name conflicts among different applications. For example, you can modify the IDL for the `Finance` module to include a `prefix` pragma as follows:

```
// IDL
# pragma prefix "USB"
module Finance {
    interface Account {
        readonly attribute float balance;
        ...
    };
    interface Bank {
        Account newAccount();
    };
};
```

These definitions yield the following repository IDs:

```
IDL:USB/Finance:1.0
IDL:USB/Finance/Account:1.0
IDL:USB/Finance/Account/balance:1.0
IDL:USB/Finance/Bank:1.0
IDL:USB/Finance/Bank/newAccount:1.0
```

Version pragma

A version number for an IDL definition's repository ID can be specified with a `version` pragma. The `version` pragma directive uses the following format:

`#pragma version` *`name major.minor`*

`name` can be a fully scoped name or an identifier whose scope is interpreted relative to the scope in which the pragma directive is included. If no version pragma is specified for an IDL definition, the default version number is 1.0. For example:

```
// IDL
module Finance {
    #pragma version Account 2.5
    interface Account {
        // ...
    };
};
```

These definitions yield the following repository IDs:

```
IDL:Finance:1.0
```

```
IDL:Finance/Account:2.5
```

Version numbers are embedded in the string format of an object reference. A client can invoke on the corresponding server object only if its interface has a matching version number, or has no version associated with it.

Note: You cannot populate the interface repository with two IDL interfaces that share the same name but have different version numbers.

CHAPTER 17

Naming Service

The Orbix naming service lets you associate names with objects. Servers can register object references by name with the naming service repository, and advertise those names to clients. Clients, in turn, can resolve the desired objects in the naming service by supplying the appropriate name.

The Orbix naming service implements the OMG COS Interoperable Naming Service, which describes how applications can map object references to names.

Benefits

Using the naming service can offer the following benefits:

- Clients can locate objects through standard names that are independent of the corresponding object references. This affords greater flexibility to developers and administrators, who can direct client requests to the most appropriate implementation. For example, you can make changes to an object's implementation or its location that are transparent to the client.
- The naming service provides a single repository for object references. Thus, application components can rely on it to obtain an application's initial references.

In this chapter

This chapter describes how to build and maintain naming graphs programmatically. It also shows how to use object groups to achieve load balancing. It contains these sections:

Many operations that are discussed here can also be executed administratively with Orbix tools. For more information about these and related configuration options, refer to the *Application Server Platform Administrator's Guide*.

Naming Service Design

Naming graph organization

The naming service is organized into a *naming graph*, which is equivalent to a directory system. A naming graph consists of one or more *naming contexts*, which correspond to directories. Each naming context contains zero or more name-reference associations, or *name bindings*, each of which refers to another node within the naming graph. A name binding can refer either to another naming context or to an object reference. Thus, any path within a naming graph finally resolves to either a naming context or an object reference. All bindings in a naming graph can usually be resolved via an *initial naming context*.

Example

Figure 21 shows how the Account interface described in earlier chapters might be extended (through inheritance) into multiple objects, and organized into a hierarchy of naming contexts. In this graph, hollow nodes are naming contexts and solid nodes are application objects. Naming contexts are typically intermediate nodes, although they can also be leaf nodes; application objects can only be leaf nodes.

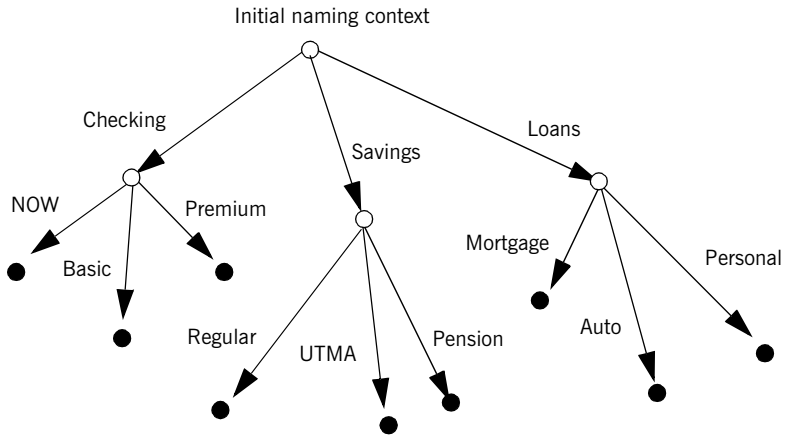


Figure 21: *A naming graph is a hierarchy of naming contexts*

Each leaf node in this naming graph associates a name with a reference to an account object such as a basic checking account or a personal loan account. Given the full path from the initial naming context—for example, `Savings/Regular`—a client can obtain the associated reference and invoke requests on it.

The operations and types that the naming service requires are defined in the IDL file `CosNaming.idl`. This file contains a single module, `CosNaming`, which in turn contains three interfaces: `NamingContext`, `NamingContextExt`, and `BindingIterator`.

Defining Names

Name sequence

A naming graph is composed of `Name` sequences of `NameComponent` structures, defined in the `CosNaming` module:

```
module CosNaming{
    typedef string Istring;
    struct NameComponent {
            Istring id;
            Istring kind;
    }
    typedef  sequence<NameComponent> Name;
    ...
};
```

A `Name` sequence specifies the path from a naming context to another naming context or application object. Each name component specifies a single node along that path.

Name components

Each name component has two string members:

- The `id` field acts as a name component's principle identifier. This field must be set.
- The `kind` member is optional; use it to further differentiate name components, if necessary.

Both `id` and `kind` members of a name component are used in name resolution. So, the naming service differentiates between two name components that have the same ids but different kinds.

For example, in the naming graph shown in Figure 21 on page 423, the path to a Personal loan account object is specified by a `Name` sequence in which only the `id` fields are set:

Figure 0.1:

Index	id	kind
0	Loans	
1	Personal	

In order to bind another Personal account object to the same Loan naming context, you must differentiate it from the existing one. You might do so by setting their `kind` fields as follows:

Figure 0.2:

Index	id	kind
0	Loans	
1	Personal	unsecured
1	Personal	secured

Note: If the `kind` field is unused, it must be set to an empty string.

Representing Names as Strings

The `CosNaming::NamingContextExt` interface defines a `StringName` type, which can represent a `Name` as a string with the following syntax:

`id[.kind][/id[.kind] ] ...`

Name components are delimited by a forward slash (`/`); `id` and `kind` members are delimited by a period (`.`). If the name component contains only the `id` string, the `kind` member is assumed to be an empty string.

`StringName` syntax reserves the use of three characters: forward slash (`/`), period (`.`), and backslash (`\`). If a name component includes these characters, you can use them in a `StringFormat` by prefixing them with a backslash (`\`) character.

The `CosNaming::NamingContextExt` interface provides several operations that allow conversion between `StringName` and `Name` data:

- `to_name()` converts a `StringName` to a `Name` (see page 428).
- `to_string()` converts a `Name` to a `StringName` (see page 430).
- `resolve_str()` uses a `StringName` to find a `Name` in a naming graph and returns an object reference (see page 440).

Note: You can invoke these and other `CosNaming::NamingContextExt` operations only on an initial naming context that is narrowed to `CosNaming::NamingContextExt`.

Initializing a Name

You can initialize a `CosNaming::Name` sequence in one of two ways:

- Set the members of each name component.
- Call `to_name()` on the initial naming context and supply a `StringName` argument. This operation converts the supplied string to a `Name` sequence.

Setting name component members

Given the loan account objects shown earlier, you can set the name for an unsecured personal loan as follows:

Example 38: *Initializing a name*

```
org.omg.CosNaming.NameComponent[] name =
    new org.omg.CosNaming.NameComponent[]
{
    new NameComponent("Loans", "");
    new nameComponent("Personal", "unsecured");
};
```

Converting a stringname to a name

The name shown in the previous example can also be set in a more straightforward way by calling `to_name()` on the initial naming context (see "Obtaining the Initial Naming Context" on page 431):

Example 39: *Using to_name() to initialize a Name*

```
// get initial naming context
org.omg.CosNaming.NamingContextExt root_cxt = ...;

org.omg.CosNaming.NameComponent[] name =
    root_cxt.to_name("Loans/Personal.unsecured");
```

The `to_name()` operation takes a string argument and returns a`CosNaming::Name`, which the previous example sets as follows:

Figure 0.3:

Index	id	kind
0	Loans	

Figure 0.3:

Index	id	kind
1	Personal	unsecured

Converting a Name to a StringName

You can convert a `CosNaming::Name` to a `CosNamingExt::StringName` by calling `to_string()` on the initial naming context. This lets server programs to advertise human-readable object names to clients.

For example, the following code converts `Name` sequence `name` to a `StringName`:

Example 40: *Converting a Name to a StringName*

```
// get initial naming context
org.omg.CosNaming.NamingContextExt root_cxt = ...;

// initialize name
org.omg.CosNaming.NameComponent[] name = ...;

...
org.omg.CosNaming.NamingContextExt.StringName str_n;
str_n = root_cxt.to_string(name);
```

Obtaining the Initial Naming Context

Clients and servers access a naming service through its initial naming context, which provides the standard entry point for building, modifying, and traversing a naming graph. To obtain the naming service's initial naming context, call `resolve_initial_references()` on the ORB. For example:

Example 41: *Obtaining the inital naming context*

```
// Initialize the ORB
global_orb = org.omg.CORBA.ORB.init(args, null);
// Get reference to initial naming context
org.omg.CORBA.Object obj =
   global_var.resolve_initial_references("NameService");
```

To obtain a reference to the naming context, narrow the result with `CosNaming.NamingContextExtHelper.narrow()`:

```
...
org.omg.CosNaming.NamingContextExt root_cxt;
if (root_cxt =
        org.omg.CosNaming.NamingContextExtHelper.narrow(obj)) {

} else {...} // Deal with failure to narrow()
...
```

A naming graph's initial naming context is equivalent to the root directory. Later sections show how you use the initial naming context to build and modify a naming graph, and to resolve names to object references.

Note: The `NamingContextExt` interface provides extra functionality over the `NamingContext` interface; therefore, the code in this chapter assumes that an initial naming context is narrowed to the `NamingContextExt` interface

Building a Naming Graph

A name binding can reference either an object reference or another naming context. By binding one naming context to another, you can organize application objects into logical categories. However complex the hierarchy, almost all paths within a naming graph hierarchy typically resolve to object references.

In an application that uses a naming service, a server program often builds a multi-tiered naming graph on startup. This process consists of two repetitive operations:

- Bind naming contexts into the desired hierarchy.
- Bind objects into the appropriate naming contexts.

Binding Naming Contexts

A server that builds a hierarchy of naming contexts contains the following steps:

1. Gets the initial naming context (see page 431).
2. Creates the first tier of naming contexts from the initial naming context.
3. Binds the new naming contexts to the initial naming context.
4. Adds naming contexts that are subordinate to the first tier:
   - Creates a naming context from any existing one.
   - Binds the new naming context to its designated parent.

The naming graph shown in Figure 21 on page 423 contains three naming contexts that are directly subordinate to the initial naming context: Checking, Loans, and Savings. The following code binds the Checking naming context to the initial naming context, as shown in Figure 22:

Example 42: *Binding a naming context to the initial naming context*

```
//get initial naming context
org.omg.CosNaming.NamingContextExt root_cxt = ...;

// create naming context
org.omg.CosNaming.NamingContext checking_cxt =
    root_cxt.new_context();

// initialize name
org.omg.CosNaming.NameComponent[] name = new NameComponent[1];
name[0] = new NameComponent("Checking", "");

// bind new context
root_cxt.bind_context(name, checking_cxt);
```

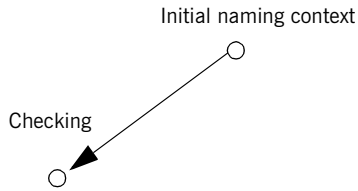


Figure 22: *Checking context bound to initial naming context*

Similarly, you can bind the Savings and Loans naming contexts to the initial naming context. The following code uses the shortcut operation `bind_new_context()`, which combines `new_context()` and `bind()`. It also uses the `to_name()` operation to set the `Name` variable.

Example 43: *Binding a naming context with bind_new_context()*

```
org.omg.CosNaming.NamingContext savings_cxt, loan_cxt;

// create naming contexts
name = root_cxt.to_name("Savings");
savings_cxt = root_cxt.bind_new_context(name);

name = root_cxt.to_name("Loan");
loan_cxt = root_cxt.bind_new_context(name);
```

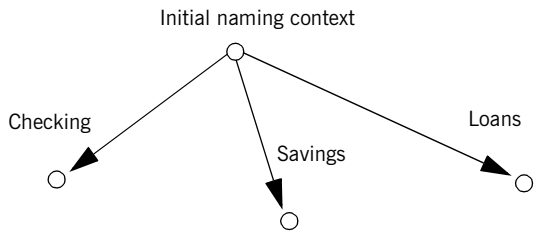


Figure 23: *Savings and Loans naming contexts bound to initial naming context*

Orphaned naming contexts

The naming service can contain naming contexts that are unbound to any other context. Because these naming contexts have no parent context, they are regarded as *orphaned*. Any naming context that you create with

new_context() is orphaned until you bind it to another context. Although it has no parent context, the initial naming context is not orphaned inasmuch as it is always accessible through resolve_initial_references(), while orphan naming contexts have no reliable means of access.

You might deliberately leave a naming context unbound—for example, you are in the process of constructing a new branch of naming contexts but wish to test it before binding it into the naming graph. Other naming contexts might appear to be orphaned within the context of the current naming service; however, they might actually be bound to a federated naming graph in another naming service (see "Federating Naming Graphs" on page 450).

Erroneous usage of orphaned naming contexts

Orphaned contexts can also occur inadvertently, often as a result of carelessly written code. For example, you can create orphaned contexts as a result of calling rebind() or rebind_context() to replace one name binding with another (see "Rebinding" on page 438). The following code shows how you might orphan the Savings naming context:

Example 44: *Orphaned naming contexts*

```
//get initial naming context
org.omg.CosNaming.NamingContextExt root_cxt = ...;

org.omg.CosNaming.NamingContext savings_cxt;

// initialize name
org.omg.CosNaming.NameComponent[] name = new NameComponent[1];
name[0] = new NameComponent("Savings", "");

// create and bind checking_cxt
savings_cxt = root_cxt.bind_new_context(name);

// make another context
org.omg.CosNaming.NamingContext savings_cxt2;
savings_cxt2 = root_cxt.new_context();

// bind savings_cxt2 to root context, savings_cxt now orphaned!
root_cxt.rebind_context(name, savings_cxt2);
```

An application can also create an orphan context by calling unbind() on a context without calling destroy() on the same context object (see "Maintaining the Naming Service" on page 448).

In both cases, if the application exits without destroying the context objects, they remain in the naming service but are inaccessible and cannot be deleted.

Binding Object References

After you construct the desired hierarchy of naming contexts, you can bind object references to them with the `bind()` operation. The following example builds on earlier code to bind a Basic checking account object to the Checking naming context:

Example 45: *Binding an object reference*

```
// object reference "basic_check" obtained earlier
...

name[0] = new NameComponent("Basic", "");
checking_cxt.bind(name, basic_check);
```

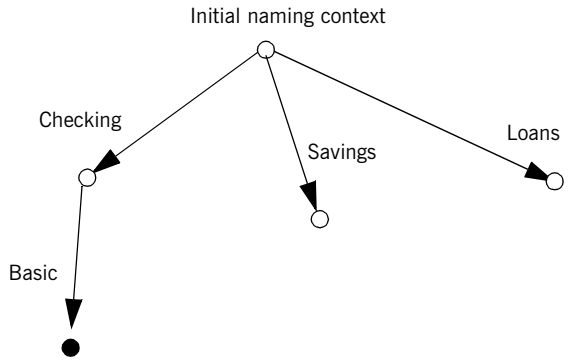


Figure 24: *Binding an object reference to a naming context*

The previous code assumes the existence of a `NamingContext` variable for the `Checking` naming context on which you can invoke `bind()`. Alternatively, you can invoke `bind()` on the initial naming context in order to bind `Basic` into the naming graph:

```
name = root_cxt.to_name("Checking/Basic");
root_cxt.bind(name, basic_check);
```

Note: Because the initial naming context is always available, it is the most reliable way to access all other contexts within a naming graph.

Rebinding

If you call `bind()` or `bind_context()` on a naming context that already contains the specified binding, the naming service throws an exception of `AlreadyBound`. To ensure the success of a binding operation whether or not the desired binding already exists, call one of the following naming context operations:

- `rebind()` rebinds an application object.
- `rebind_context()` rebinds a naming context.

Either operation replaces an existing binding of the same name with the new binding. Calls to `rebind()` in particular can be useful on server startup, to ensure that the naming service has the latest object references.

Note: Calls to `rebind_context()` or `rebind()` can have the undesired effect of creating orphaned naming contexts (see page 434). In general, exercise caution when calling either function.

Using Names to Access Objects

A client application can use the naming service to obtain object references in three steps:

1. Obtain a reference to the initial naming context (see page 431).
2. Set a CosNaming::Name structure with the full path of the name associated with the desired object.
3. Resolve the name to the desired object reference.

Setting object names

You specify the path to the desired object reference in a `CosNaming::Name`. You can set this name in one of two ways:

Explicitly set the `id` and `kind` members of each `Name` element. For example, the following code sets the name of a Basic checking account object:

Example 46: *Setting object name components*

```
org.omg.CosNaming.NameComponent[] name =
    new NameComponent[2];
name[0] = new NameComponent("Checking", "");
name[1] = new NameComponent("Basic", "");
```

Call `to_name()` on the initial naming context. This option is available if the client code narrows the initial naming context to the `NamingContextExt` interface. `to_name()` takes a `CosNaming::CosNamingExt::StringName` argument and returns a `CosNaming::Name` as follows:

Example 47: *Setting an object name with to_name()*

```
org.omg.CosNaming.NameComponent[] name =
    root_cxt.to_name("Checking/Basic");
```

For more about using a `StringName` with `to_name()`, see "Converting a stringname to a name" on page 428.

Resolving names

Clients call `resolve()` on the initial naming context to obtain the object associated with the supplied name:

Example 48: *Calling resolve()*

```
org.omg.CORBA.Object obj;
...
obj = root_cxt.resolve(name);
```

Alternatively, the client can call `resolve_str()` on the initial naming context to resolve the same name using its `StringName` equivalent:

Example 49: *Calling resolve_str()*

```
org.omg.CORBA.Object obj;
...
obj = root_cxt.resolve_str("Checking/Basic");
```

In both cases, the object returned in `obj` is an application object that implements the IDL interface `BasicChecking`, so the client narrows the returned object accordingly:

```
BasicChecking checking;
...
try {
    checking = BasicCheckingHelper.narrow(obj);
    // perform some operation on basic checking object
    ...
} // end of try clause, catch clauses not shown
```

Resolving names with corbaname

You can resolve names with a `corbaname` URL, which is similar to a `corbaloc` URL (see "Using corbaloc URL strings" on page 244). However, a `corbaname` URL also contains a stringified name that identifies a binding in a naming context. For example, the following code uses a corbaname URL to obtain a reference to a BasicChecking object:

Example 50: *Resolving a name with corbaname*

```
org.omg.CORBA.Object obj;
obj = orb.string_to_object(
     "corbaname:rir:/NameService#Checking/Basic"
);
```

A corbaname URL has the following syntax:

`corbaname:rir:`*`[/NameService]#string-name`*

`string-name` is a string that conforms to the format allowed by a `CosNaming::CosNamingExt::StringName` (see "Representing Names as Strings" on page 427). A corbaname can omit the `NameService` specifier. For example, the following call to `string_to_object()` is equivalent to the call shown earlier:

```
obj = orb.string_to_object("corbaname:rir:#Checking/Basic");
```

Exceptions Returned to Clients

Invocations on the naming service can result in the following exceptions:

NotFound The specified name does not resolve to an existing binding. This exception contains two data members:

`why` Explains why a lookup failed with one of the following values:

- `missing_node`: one of the name components specifies a non-existent binding.
- `not_context`: one of the intermediate name components specifies a binding to an application object instead of a naming context.
- `not_object`: one of the name components points to a non-existent object.

`rest_of_name` Contains the trailing part of the name that could not be resolved.

InvalidName The specified name is empty or contains invalid characters.

CannotProceed The operation fails for reasons not described by other exceptions. For example, the naming service's internal repository might be in an inconsistent state.

AlreadyBound Attempts to create a binding in a context throw this exception if the context already contains a binding of the same name.

Not Empty Attempts to delete a context that contains bindings throw this exception. Contexts must be empty before you delete them.

Listing Naming Context Bindings

In order to find an object reference, a client might need to iterate over the bindings in one or more naming contexts. You can invoke the `list()` operation on a naming context to obtain a list of its name bindings. This operation has the following signature:

```
void list(
    in unsigned long how_many,
    out BindingList bl,
    out BindingIterator it);
```

`list()` returns with a `BindingList`, which is a sequence of `Binding` structures:

```
enum BindingType{ nobject, ncontext };

struct Binding{
    Name binding_name
    BindingType binding_type;
}
typedef sequence<Binding> BindingList
```

Iterating over binding list elements

Given a binding list, the client can iterate over its elements to obtain their binding name and type. Given a `Binding` element's name, the client application can call `resolve()` to obtain an object reference; it can use the binding type information to determine whether the object is a naming context or an application object.

For example, given the naming graph in Figure 21, a client application can invoke `list()` on the initial naming context and return a binding list with three `Binding` elements:

Figure 0.4:

Index	Name	BindingType
0	Checking	ncontext
1	Savings	ncontext

Figure 0.4:

Index	Name	BindingType
2	Loan	ncontext

Using a Binding Iterator

Limiting number of bindings returned by list()

In the previous example, `list()` returns a small binding list. However, an enterprise application is likely to require naming contexts with a large number of bindings. `list()` therefore provides two parameters that let a client obtain all bindings from a naming context without overrunning available memory:

how_many sets the maximum number of elements to return in the binding list. If the number of bindings in a naming context is greater than `how_many`, `list()` returns with its `BindingIterator` parameter set.

it is a `BindingIterator` object that can be used to retrieve the remaining bindings in a naming context. If `list()` returns with all bindings in its `BindingList`, this parameter is set to nil.

A `BindingIterator` object has the following IDL interface definition:

```
interface BindingIterator{
    boolean next_one(out Binding b);
    boolean next_n(in unsigned long how_many, out BindingList
   bl);
    void destroy();
}
```

Obtaining remainder of bindings

If `list()` returns with a `BindingIterator` object, the client can invoke on it either `next_n()` to retrieve the next specified number of remaining bindings, or `next_one()` to retrieve one remaining binding at a time. Both functions return true if the naming context contains more bindings to fetch. Together, these `BindingIterator` operations and `list()` let a client safely obtain all bindings in a context.

Note: The client is responsible for destroying an iterator. It also must be able to handle exceptions that might return when it calls an iterator operation, inasmuch as the naming service can destroy an iterator at any time before the client retrieves all naming context bindings.

The following client code gets a binding list from a naming context and prints each element's binding name and type:

Example 51: *Obtaining a binding list*

```
// printing function
void
print_binding_list(org.omg.CosNaming.BindingListHolder bl)
{
    // extract the list of bindings
    org.omg.CosNaming.Binding[] list = bl.value;
    // iterate through list
    for( int i = 0; i < list.length; i++ ){
        System.out.print( list[i].binding_name[0].id;
        if( list[i].binding_name[0].kind != null )
            System.out.print(
                "(" + bl[i].binding_name[0].kind + ")");
        if( bl[i].binding_type ==
            org.omg.CosNaming.BindingType.ncontext )
            System.out.println( ": naming context" );
        else
            System.out.println( ": object reference" );
    }
}

void
get_context_bindings(omg.org.CosNaming.NamingContext cxt)
{
    org.omg.CosNaming.BindingListHolder b_list;
    org.omg.CosNaming.BindingIteratorHolder b_iter =
        new org.omg.CosNaming.BindingIteratorHolder();
    long MAX_BINDINGS = 50;

    // set up array to store binding list, put it in holder
    org.omg.CosNaming.Binding[] binding_list =
        new org.omg.CosNaming.Binding[MAX_BINDINGS];
    b_list =
        new org.omg.CosNaming.BindingListHolder(binding_list);

    // get first set of bindings from cxt
    cxt.list(MAX_BINDINGS, b_list, b_iter);
```

Example 51: *Obtaining a binding list*

```
    //print first set of bindings
    print_binding_list(b_list);

    // look for remaining bindings
    if( b_iter.value != null ) {
        org.omg.CosNaming.BindingIterator it = b_iter.value;
        do {
            boolean more = it.next_n(MAX_BINDINGS, b_list);
            // print next set of bindings
            print_binding_list(b_list);
        } while (more);

    // get rid of iterator
    it.destroy();
    }
}
```

When you run this code on the initial naming context shown earlier, it yields the following output:

```
Checking: naming context
Savings: naming context
Loan: naming context
```

Maintaining the Naming Service

Destruction of a context and its bindings is a two-step procedure:

- Remove bindings to the target context from its parent contexts by calling `unbind()` on them.
- Destroy the context by calling the `destroy()` operation on it. If the context contains bindings, these must be destroyed first; otherwise, `destroy()` returns with a `NotEmpty` exception.

These operations can be called in any order; but it is important to call both. If you remove the bindings to a context without destroying it, you leave an orphaned context within the naming graph that might be impossible to access and destroy later (see "Orphaned naming contexts" on page 434). If you destroy a context but do not remove its bindings to other contexts, you leave behind bindings that point nowhere, or *dangling bindings*.

For example, given the partial naming graph in Figure 25, you can destroy the Loans context and its bindings to the loan account objects as follows:

Example 52: *Destroying a naming context*

```
org.omg.CosNaming.NameComponent[] name;

// get initial naming context
org.omg.CosNaming.NamingContextExt root_cxt = ...;

// assume availability of Loans naming context variable
org.omg.CosNaming.NamingContext loans_cxt = ... ;

// remove bindings to Loans context
name = root_cxt.to_name("Loans/Mortgage");
root_cxt.unbind(name);
name = root_cxt.to_name("Loans/Auto");
root_cxt.unbind(name);
name = root_cxt.to_name("Loans/Personal");
root_cxt.unbind(name);

// remove binding from Loans context to initial naming context
name = root_cxt.to_name("Loans");
root_cxt.unbind(name);
```

Example 52: *Destroying a naming context*

```
// destroy orphaned Loans context
loans_cxt.destroy();
```

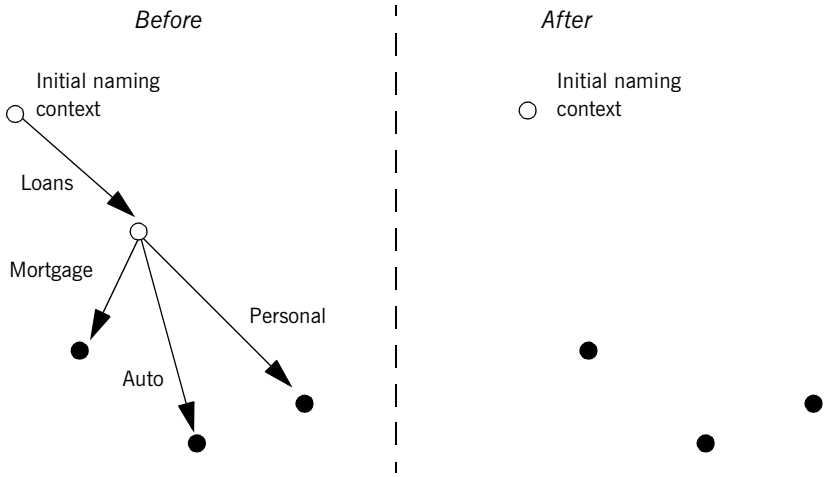


Figure 25: *Destroying a naming context and removing related bindings*

Note: Orbix provides administrative tools to destroy contexts and remove bindings. These are described in the *Application Server Platform Administrator's Guide*.

Federating Naming Graphs

A naming graph can span multiple naming services, which can themselves reside on different hosts. Given the initial naming context of an external naming service, a naming context can transparently bind itself to that naming service's naming graph. A naming graph that spans multiple naming services is said to be *federated*.

Benefits

A federated naming graph offers the following benefits:

- *Reliability*: By spanning a naming graph across multiple servers, you can minimize the impact of a single server's failure.
- *Load balancing*: You can distribute processing according to logical groups. Multiple servers can share the work load of resolving bindings for different clients.
- *Scalability*: Persistent storage for a naming graph is spread across multiple servers.
- *Decentralized administration*: Logical groups within a naming graph can be maintained separately through different administrative domains, while they are collectively visible to all clients across the network.

Federation models

Each naming graph in a federation must obtain the initial naming context of other members in order to bind itself to them. The binding possibilities are virtually infinite; however, two federation models are widely used:

- Hierarchal federation — All naming graphs are bound to a root server's naming graph. Clients access objects via the initial naming context of the root server.
- Fully-connected federation — Each naming graph directly binds itself to all other naming graphs. Typically, each naming graph binds the initial naming contexts of all other naming graphs into its own initial naming context. Clients can access all objects via the initial naming context of their local naming service.

Hierarchal federation

Figure 26 shows a hierarchal naming service federation that comprises three servers. The Deposits server maintains naming contexts for checking and savings accounts, while the Loans server maintains naming contexts for loan accounts. A single root server serves as the logical starting point for all naming contexts.

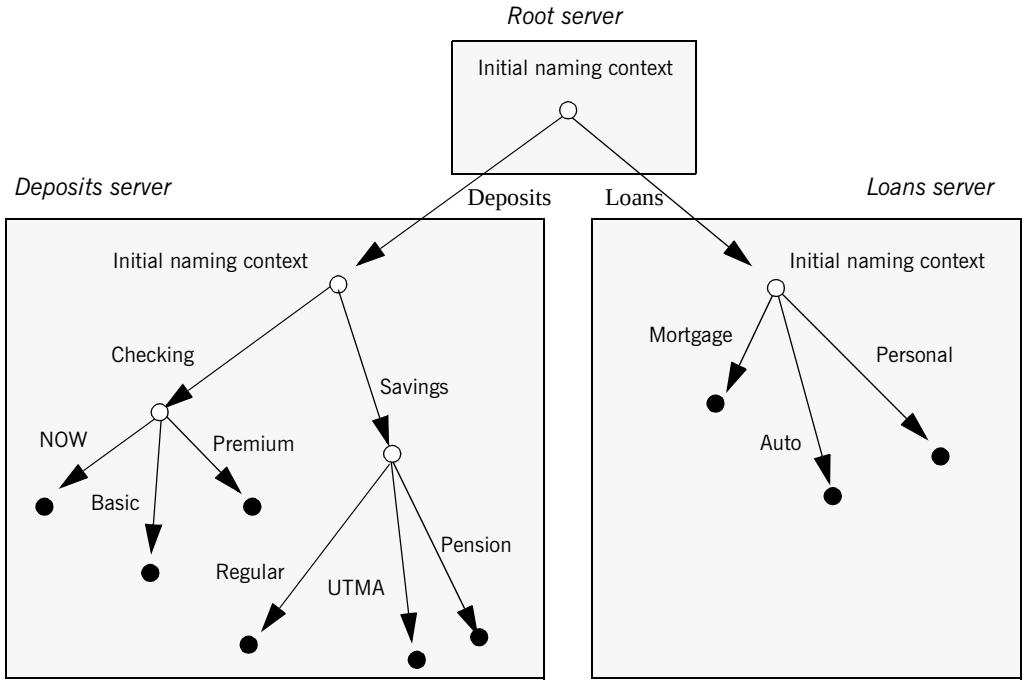


Figure 26: *A naming graph that spans multiple servers*

In this hierarchical structure, the naming graphs in the Deposits and Loans servers are federated through an intermediary root server. The initial naming contexts of the Deposits and Loans servers are bound to the root server's initial naming context. Thus, clients gain access to either naming graph through the root server's initial naming context.

The following code binds the initial naming contexts of the Deposits and Loans servers to the root server's initial naming context:

Example 53: *Federating naming graphs to a root server's initial naming context*

```
// Root server
...
public static void main (String[] args) {
    org.omg.CosNaming.NamingContextExt
        root_inc, deposits_inc, loans,_inc;
    org.omg.CosNaming.NameComponent[] name = new
   NameComponent[1];
    org.omg.CORBA.Object obj;
    org.omg.CORBA.ORB global_orb;
    String loans_inc_ior, deposits_inc_ior
...
    try {
        global_orb = org.omg.CORBA.global_orb.init(args, null);

        // code to obtain stringified IORs of initial naming
        // contexts for Loans and Deposits servers (not shown)
        ...

        obj = global_orb.string_to_object(loans_inc_ior);
        loans_inc =
            org.omg.CosNaming.NamingContextExtHelper.narrow(obj);
        obj = global_orb.string_to_object(deposits_inc_ior);
        deposits_inc =
            org.omg.CosNaming.NamingContextExtHelper.narrow(obj);

        // get initial naming context for Root server
        root_inc = ... ;

        // bind Deposits initial naming context to root server's
        // initial naming context
        name[0] = new NameComponent("Deposits", "");
        root_inc.bind_context(name, deposits_inc);

        // bind Loans initial naming context to root server's
        // initial naming context
        name[0] = new NameComponent("Loans", "");
        root_inc.bind_context(name, deposits_inc);
    }
}
```

This yields the following bindings between the three naming graphs:

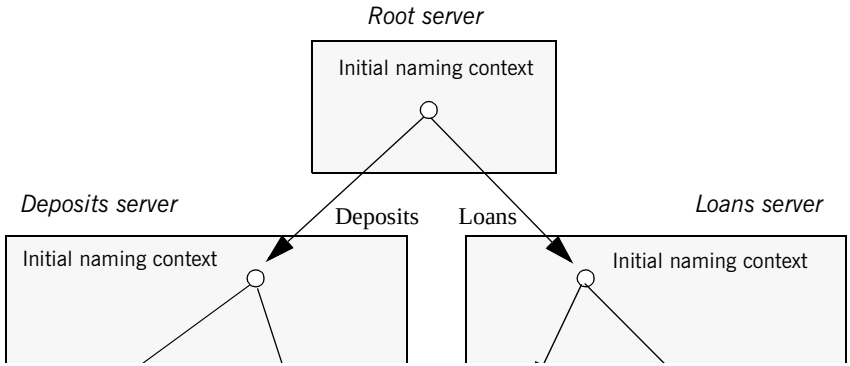


Figure 27: *Multiple naming graphs are linked by binding initial naming contexts of several servers to a root server.*

Fully-connected federation

In a purely hierarchical model like the naming graph just shown, clients obtain their initial naming context from the root server, and the root server acts as the sole gateway into all federated naming services. To avoid bottlenecks, it is possible to modify this model so that clients can gain access to a federated naming graph via the initial naming context of any member naming service.

The next code example shows how the Deposits and Loans servers can bind the root server's initial naming context into their respective initial naming contexts. Clients can use this binding to locate the root server's initial naming context, and then use root-relative names to locate objects.

Figure 28 shows how this federates the three naming graphs:

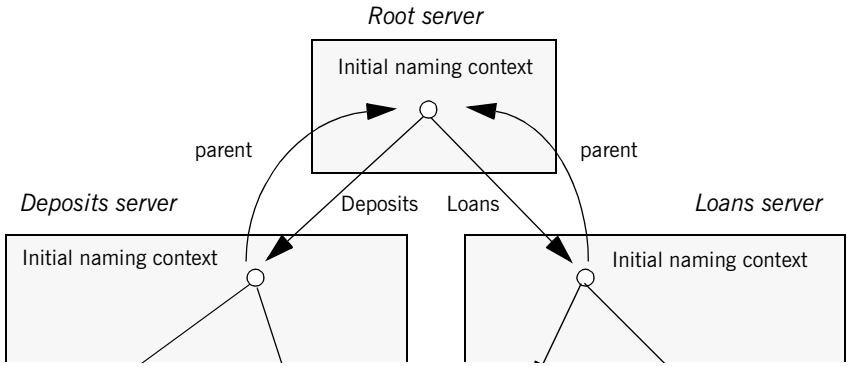


Figure 28: *The root server's initial naming context is bound to the initial naming contexts of other servers, allowing clients to locate the root naming context.*

The code for both Deposits and Loans server processes is virtually identical:

Example 54: *Federating naming graphs through the initial naming contexts of multiple servers*

```
public static void main (String[] args) {
    org.omg.CosNaming.NamingContextExt root_inc, this_inc;
    org.omg.CosNaming.NameComponent[] name =
        new NameComponent[1];
    org.omg.CORBA.Object obj;
    org.omg.CORBA.ORB global_orb;
    String root_inc_ior;
    ...
    try {
        global_orb = org.omg.CORBA.global_orb.init(args, null);

        // code to obtain stringified IORs of root server's
        // initial naming context (not shown)
        ...

        obj = global_orb.string_to_object(root_inc_ior);
        root_inc =
            org.omg.CosNaming.NamingContextExtHelper.narrow(obj);
```

Example 54: *Federating naming graphs through the initial naming contexts of multiple servers*

```
        // get initial naming context for this server
        this_inc = ... ;

        name[0] = new NameComponent("parent", "");

        // bind root server's initial naming context to
        // this server's initial naming context
        this_inc.bind_context(name, root_inc);
        ...
    }
```

Sample Code

The following sections show the server and client code that is discussed in previous sections of this chapter.

Server code

Example 55: *Server naming service code*

```
public static void main (String[] args) {
    org.omg.CosNaming.NamingContextExt root_cxt;
    org.omg.CosNaming.NamingContext
        checking_cxt, savings_cxt, loan_cxt;
    org.omg.CosNaming.NameComponent[] name;
    org.omg.CORBA.ORB orb;
    org.omg.CORBA.Object obj;
    Checking basic_check, now_check, premium_check;

    // Checking objects initialized from persistent data
    // (not shown)

    try {
        // Initialize the ORB
        orb = org.omg.CORBA.global_orb.init(args, null);

        // Get reference to initial naming context
        obj =
            global_orb.resolve_initial_references("NameService");
        root_cxt =
            org.omg.CosNaming.NamingContextExtHelper.narrow(obj);
        if( root_cxt != null ) {
            // build naming graph

            // initialize name
            name = root_cxt.to_name("Checking");
            // bind new naming context to root
            checking_cxt = root_cxt.bind_new_context(name);
```

Example 55: *Server naming service code*

```
            // bind checking objects to Checking context
            name = root_cxt.to_name("Checking/Basic");
            checking_cxt.bind(name, basic_check);
            name = root_cxt.to_name("Checking/Premium");
            checking_cxt.bind(name, premium_check);
            name = root_cxt.to_name("Checking/NOW");
            checking_cxt.bind(name, now_check);

            name = root_cxt.to_name("Savings");
            savings_cxt = root_cxt.bind_new_context(name);

            // bind savings objects to savings context
            ...

            name = root_cxt.to_name("Loan");
            loan_cxt = root_cxt.bind_new_context(name);

            // bind loan objects to loan context
            ...
        }
        else {...} // deal with failure to narrow()
        ...
    } // end of try clause, catch clauses not shown
    ...
}
```

Client code

Example 56: *Client naming service code*

```
public static void main (String[] args) {
    org.omg.CosNaming.NamingContextExt root_cxt;
    org.omg.CosNaming.NameComponent[] name;
    BasicChecking_var checking;
    org.omg.CORBA.Object obj;
    org.omg.CORBA.ORB global_orb;
...

    try {
        global_orb = org.omg.CORBA.global_orb.init (args, null);
```

Example 56: *Client naming service code*

```
        // Find the initial naming context
        obj =
           global_orb.resolve_initial_references("NameService");
        root_cxt =
          org.omg.CosNaming.NamingContextExtHelper.narrow(obj);
        if( root_cxt != null ) {
            obj = root_cxt.resolve_str("Checking/Basic");
            checking_var == BasicCheckingHelper.narrow(obj);
            if( checking_var != null ) {
               // perform some operation on basic checking object
                ...
            }
            else { ... } // Deal with failure to narrow()
        } else { ... } // Deal with failure to resolve object

    } // end of try clause, catch clauses not shown
    ...
}
```

Object Groups and Load Balancing

The naming service defines a repository of names that map to objects. A name maps to one object only. Orbix extends the naming service model to allow a name to map to a group of objects. An *object group* is a collection of objects that can increase or decrease in size dynamically.

Selection algorithms

Each object group has a selection algorithm that is set when the object group is created (see page 463). This algorithm is applied when a client resolves the name associated with the object group; and the naming service directs client requests to objects accordingly.

Three selection algorithms are supported:

Round-robin: The locator uses a round-robin algorithm to select from the list of active servers—that is, the first client is sent to the first server, the second client to the second server, and so on.

Random: The locator randomly selects an active server to handle the client.

Active load balancing: Each object group member is assigned a load value. The naming service satisfies client resolve() invocations by returning references to members with the lowest load values.

Figure 29 shows how a name can bind to multiple objects through an object group.

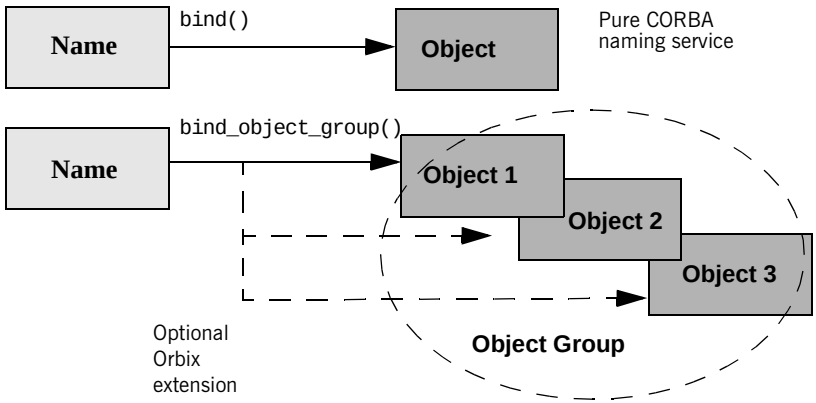


Figure 29: *Associating a name with an object group*

Orbix supports object groups through its own IDL interfaces. These interfaces let you create object groups and manipulate them: add objects to and remove objects from groups, and find out which objects are members of a particular group. Object groups are transparent to clients.

Load balancing interfaces

IDL modules `IT_LoadBalancing` and `IT_Naming`, defined in `orbix/load_balancing.idl` and `orbix/naming.idl`, respectively, provide operations that allow access to Orbix load balancing:

```
module IT_LoadBalancing
{
    exception NoSuchMember{};
    exception DuplicateMember{};
    exception DuplicateGroup{};
    exception NoSuchGroup{};
```

```
typedef string MemberId;
typedef sequence<MemberId> MemberIdList;

enum SelectionMethod
    { ROUND_ROBIN_METHOD, RANDOM_METHOD, ACTIVE_METHOD };

struct Member
{
    Object obj;
    MemberId id;
};

typedef string GroupId;
typedef sequence<GroupId> GroupList;

interface ObjectGroup
{
    readonly attribute string id;
    attribute SelectionMethod selection_method;
    Object pick();
    void add_member (in Member mem)
        raises (DuplicateMember);
    void remove_member (in MemberId id)
        raises (NoSuchMember);
    Object get_member (in MemberId id)
        raises (NoSuchMember);
    MemberIdList members();
    void destroy();
    void update_member_load(
        in MemberIdList ids,
        in double curr_load
    )  raises (NoSuchMember);
    double get_member_load(
        in MemberId id
    )  raises (NoSuchMember);
    void set_member_timeout(
        in MemberIdList ids,
        in long timeout_sec
    )  raises (NoSuchMember);
    long get_member_timeout(
        in MemberId id
    )  raises (NoSuchMember);
};
```

```
    interface ObjectGroupFactory
    {
        ObjectGroup create_round_robin (in GroupId id)
            raises (DuplicateGroup);
        ObjectGroup create_random (in GroupId id)
            raises (DuplicateGroup);
        ObjectGroup create_active (in GroupId id)
            raises (DuplicateGroup);
        ObjectGroup find_group (in GroupId id)
            raises (NoSuchGroup);
        GroupList rr_groups();
        GroupList random_groups();
        GroupList active_groups();
    };
};
```

For detailed information about these interfaces, see the *CORBA Programmer's Reference*.

Using Object Groups in Orbix

The `IT_LoadBalancing` module lets servers perform the following tasks:

- Create an object group and add objects to it.
- Add objects to an existing object group.
- Remove objects from an object group.
- Remove an object group.
- Set member load values and direct client requests accordingly.

Create an object group

You create an object group and add objects to it in the following steps:

1. Get a reference to a naming context such as the initial naming context and narrow to `IT_NamingContextExt`.
2. Create an object group factory by calling `og_factory()` on the naming context object. This returns a reference to an `IT_LoadBalancing::ObjectGroupFactory` object.
3. Create an object group by calling `create_random()`, `create_round_robin()`, or `create_active()` on the object group factory. These operations return a reference to an object group of interface `IT_LoadBalancing::ObjectGroup` that uses the desired selection algorithm.
4. Add application objects to the newly created object group by calling `add_member()` on it.
5. Bind a name to the object group by calling `bind_object_group()` on the naming context object created in step **1**.

When you create the object group, you must supply a group identifier. This identifier is a string value that is unique among other object groups.

Similarly, when you add a member to the object group, you must supply a reference to the object and a corresponding member identifier. This identifier is a string value that must be unique within the object group.

In both cases, you decide the format of the identifier string. Orbix does not interpret these identifiers.

Add objects to an existing object group

Before you add objects to an existing object group, you must get a reference to the corresponding `IT_LoadBalancing::ObjectGroup` object. You can do this by using either the group identifier or the name that is bound to the object group. This section uses the group identifier.

To add objects to an existing object group:

1. Get a reference to a naming context such as the initial naming context.
2. Narrow the reference to `IT_NamingContextExt`.
3. Call `og_factory()` on the naming context object. This returns a reference to an `ObjectGroupFactory` object.
4. Call `find_group()` on the object group factory, passing the identifier for the group as a parameter. This returns a reference to the object group.
5. Add application objects to the object group by calling `add_member()` on it.

Remove objects from an object group

Removing an object from a group is straightforward if you know the object group identifier and the member identifier for the object:

1. Get a reference to a naming context such as the initial naming context and narrow to `IT_NamingContextExt`.
2. Call `og_factory()` on the naming context object. This returns a reference to an `ObjectGroupFactory` object.
3. On the object group factory, call `find_group()`, passing the identifier for the target object group as a parameter. This operation returns a reference to the object group.
4. Call `remove_member()` on the object group to remove the required object from the group. You must specify the member identifier for the object as a parameter to this operation.

If you already have a reference to the object group, the first three steps are unnecessary.

Remove an object group

To remove an object group for which you have no reference:

1. Call `unbind()` on the initial naming context to unbind the name associated with the object group.
2. Call `og_factory()` on the initial naming context object. This returns a reference to an `ObjectGroupFactory` object.
3. Call `find_group()` on the object group factory, passing the identifier for the target object group as a parameter. This operation returns a reference to the object group.
4. Call `destroy()` on the object group to remove it from the naming service.

If you already have a reference to the target object group, steps **2** and **3** are unnecessary.

Set member load values

In an object group that uses active load balancing, each object group member is assigned a load value. The naming service satisfies client `resolve()` invocations by returning references to members with the lowest load values.

A member's default load value can be set administratively through the configuration variable `plugins:naming:lb_default_initial_load`. Thereafter, load counts should be updated with periodic calls to `ObjectGroup::update_member_load()`. `itadmin` provides an equivalent command, `nsog update_member_load`, in cases where manual intervention is required, or scripting is feasible.

You should also set or modify member timeouts with `ObjectGroup::set_member_timeout()` or with `itadmin nsog set_member_timeout`. You can configure default timeout values with the configuration variable `plugins:naming:lb_default_load_timeout`. If an object's load value is not updated within its timeout interval, its object reference becomes unavailable to client `resolve()` invocations. This typically happens because the object itself or an associated process is no longer running, and therefore cannot update the object's load value.

A member reference can be made available again to client `resolve()` invocations by resetting its load value with `ObjectGroup::update_member_load()` or `itadmin nsog update_member_load`. In general, an object's timeout should be set to an interval greater than the frequency of load count updates.

Load Balancing Example

This section uses a simple stock market system to show how to use object groups in CORBA applications. In this example, a CORBA object has access to all current stock prices. Clients request stock prices from this CORBA object and display those prices to the end user.

A realistic stock market application needs to make available many stock prices, and provide many clients with price updates immediately. Given such a high processing load, one CORBA object might be unable to satisfy client requirements. You can solve this problem by replicating the CORBA object, invisibly to the client, through object groups.

Figure 30 shows the architecture for the stock market system, where a single server creates two CORBA objects from the same interface. These objects process client requests for stock price information.

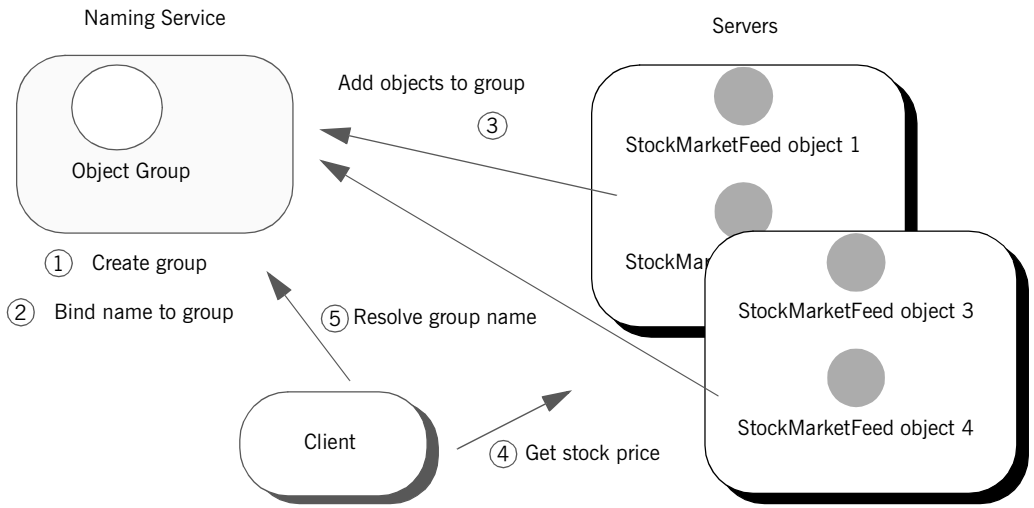


Figure 30: *Architecture of the stock market example*

Defining the IDL for the application

The IDL for the load balancing example consists of a single interface `StockMarketFeed`, which is defined in module `ObjectGroupDemo`:

```
// IDL
module ObjectGroupDemo
{
    exception StockSymbolNotFound{};
    interface StockMarketFeed
    {
        double read_stock (in string stock_symbol)
            raises(StockSymbolNotfound);
    };
};
```

`StockMarketFeed` has one operation, `read_stock()`. This operation returns the current price of the stock associated with string identifier `stock_name`, which identifies the desired stock.

Creating an Object Group and Adding Objects

After you define the IDL, you can implement the interfaces. Using object groups has no effect on how you do this, so this section assumes that you define class `StockMarketFeedServant`, which implements interface `StockMarketFeed`.

After you implement the IDL interfaces, you develop a server program that contains and manages implementation objects. The application can have one or more servers that perform these tasks:

- Creates two `StockMarketFeed` implementation objects.
- Creates an object group in the naming service.
- Adds the implementation objects to this group.

The server's `main()` routine can be written as follows:

Example 57: *Load balancing server*

```
org.omg.CORBA.ORB global_orb;
org.omg.PortableServer.POA the_poa;
String id1, id2;

public static void main (String[] args) {

    com.iona.IT_LoadBalancing.ObjectGroup rr_og_var;
    com.iona.IT_Naming.IT_NamingContextExt it_ins_var;

    // Initialize the ORB

    try {
        global_orb = org.omg.CORBA.global_orb.init(args, null);
    }
    catch (Exception ex) {
        System.out.println("Could not initialize the ORB");
        System.out.println("Exception info: "+ ex);
        System.exit(1);
    }

    // Get server name
    String server_name = (args[0]);
```

Example 57: *Load balancing server*

```
    // Initialize the POA and POA Manager
    org.omg.PortableServer.POAManager poa_manager;
    try {
        org.omg.CORBA.Object poa_obj =
            global_orb.resolve_initial_references("RootPOA");
        the_poa =
            org.omg.PortableServer.POAHelper.narrow(poa_obj);
        poa_manager = the_poa.the_POAManager();
    }
    catch (Exception ex) {
        System.out.println("Cannot obtain root POA or
   POAManager");
        System.out.println("Exception info: " + ex );
        System.exit(1);
    }

1   // Create 2 stock object servants called
    // <server_name>:RR_Member1 and <server_name>:RR_Member2:
    id1 = server_name + ":RR_Member1";
    id2 = server_name + ":RR_Member2";

    StockServantFeedServant stk_svnt1 =
        new StockServantFeedServant(id1);
    StockServantFeedServant stk_svnt2 =
        new StockServantFeedServant(id2);

2   // Get initial naming context
    com.iona.IT_LoadBalancing.ObjectGroupFactory ogf_var;
    org.omg.CORBA.Object ins_obj;
    try {
        ins_obj =
           global_orb.resolve_initial_references("NameService");
        it_ins_var =
            com.iona.IT_Naming.IT_NamingContextExtHelper.narrow
            (ins_obj);
3       ogf_var = it_ins_var.og_factory();
    }
    catch (Exception ex) {
        System.out.println("Could not narrow reference to
             IT_NamingContextExt interface. Is the Naming Service
             Running?");
        System.out.println("Exception info: " + ex );
        System.exit(1);
    }
```

Example 57: *Load balancing server*

```
    // Create a round robin object group and bind it in
    // the naming service
    String rr_id_str = "StockFeedGroup";
    try {
4       rr_og_var = ogf_var.create_round_robin(rr_id_str);
        org.omg.CosNaming.NameComponent[] nm =
            it_ins_var.to_name("stock_svc");
5       it_ins_var.bind_object_group(nm, rr_og_var);
    }
    catch (Exception ex) {
        // OK: assume other server created ObjectGroup and
        // bound it in NS
        rr_og_var = ogf_var.find_group(rr_id_str);
    }

    // Add StockMarketFeed objects to the object group
6   try
    {
        com.iona.IT_LoadBalancing.member member_info;

        member_info.id = id1;
        member_info.obj = stk_svnt1._this();
        rr_og_var.add_member(member_info);

        member_info.id = id2;
        member_info.obj = stk_svnt2._this();
        rr_og_var.add_member(member_info);
    }
    catch (Exception ex) {
    {
        System.out.println("Cannot add members " + id1
             + " , " + id2);
        System.out.println("Exception info: " + ex);
        System.exit(1);
    }

    // Start accepting requests
    try {
        poa_manager.activate();
        System.out.println ("Server ready...");
```

Example 57: *Load balancing server*

```
7           global_orb.run();
        }
        catch (Exception ex) {
            System.out.println("Unable to activate the POAManager,
                or orb.run() failed.");
            System.out.println("Exception info: " + ex );
            System.exit(1);
        }
}
```

This server executes as follows:

1. Instantiates two `StockServantFeedServant` servants that implement the `StockMarketFeed` interface.
2. Obtains a reference to the initial naming context and narrows it to `IT_Naming::IT_NamingContextExt`.
3. Obtains an object group factory by calling `og_factory()` on the naming context.
4. Calls `create_round_robin()` on the object group factory to create a new group with the specified identifier. `create_round_robin()` returns a new object group in which objects are selected on a round-robin basis.
5. Calls `bind_object_group()` on the naming context and binds a specified naming service name to this group. When a client resolves this name, it receives a reference to one of the group's member objects, selected by the naming service in accordance with the group selection algorithm.

   The enclosing `try` block should allow for the possibility that the group already exists, where `bind_object_group()` throws an exception of `CosNaming::NamingContext::AlreadyBound`. In this case, the `catch` clause calls `find_group()` in order to obtain the desired object group. `find_group()` is also useful in a distributed system, where objects must be added to an existing object group.
6. Activates two `StockMarketFeed` objects in the POA and adds them as members to the object group:

- The server creates an IDL `struct` of type `IT_LoadBalancing::member`, and initializes its two members: a string that identifies the object within the group; and a `StockMarketFeed` object reference, created by invoking `_this()` on each servant.
- The server adds the new member to the object group by calling `add_member()`.

7. Prepares to receive client requests by calling `run()` on the ORB.

Accessing Objects from a Client

All objects in an object group provide the same service to clients. A client that resolves a name in the naming service does not know whether the name is bound to an object group or a single object. The client receives a reference to one object only. A client program resolves an object group name just as it resolves a name bound to one object, using standard CORBA-compliant interfaces.

For example, the stock market client's `main()` routine might look like this:

Example 58: *Accessing objects from an object group*

```
org.omg.CORBA.ORB global_orb;

public static void main (String[] args) {

    org.omg.CosNaming.NamingContextExt ins;

    try {
        global_orb = org.omg.CORBA.global_orb.init(args, null);

        org.omg.CORBA.Object ins_obj =
            global_orb.resolve_initial_references("NameService");
        ins =
        org.omg.CosNaming.NamingContextExtHelper.narrow(ins_obj);
    }
    catch (Exception ex) {
        System.out.println(
            "Cannot resolve/narrow the name service IOR);
        System.out.println("Exception info: " + ex);
        System.exit(1);
    }
```

Example 58: *Accessing objects from an object group*

```
    StockMarketFeed stk_ref;
    try {
        org.omg.CORBA.Object stk_obj =
            ins.resolve_str("stock_svc");
        stk_ref = StockMarketFeedHelper.narrow(stk_obj);
    }
    catch (Exception ex) {
        System.out.println("Unable to resolve/narrow stock_svc
            IOR from naming service");
        System.out.println("Exception info: " + ex);
        System.exit(1);
    }

    double curr_price;
    try {
        curr_price = stk_ref.read_stock(args[0]);
    }
    catch (Exception ex) {
        System.out.println("Stock symbol not found: " + args[0]);
        System.out.println("Try another stock symbol");
        System.exit(1);
    }

    System.out.println(args[0] + " stock price is " +
   curr_price);
}
```

CHAPTER 18

Event Service

The event service enables decoupled communication between client consumers and suppliers by forwarding messages through an event channel.

An event originates at a client *supplier* and is forwarded through an *event channel* to any number of client *consumers*. Suppliers and consumers are completely decoupled: a supplier has no knowledge of the number of consumers or their identities, and consumers have no knowledge of which supplier generated a given event.

In this chapter

This chapter discusses the following topics:

Overview

Service capabilities

An event channel provides the following capabilities for forwarding events:

- Enables consumers to subscribe to events of certain types.
- Accepts incoming events from client suppliers.
- Forwards supplier-generated events to all connected consumers.
- Forwarding messages using well defined IDL interfaces.

Connections

Suppliers and consumers connect to an event channel and not directly to each other, as shown in Figure 31. From a supplier's perspective, the event channel appears as a single consumer; from a consumer's perspective, the event channel appears as a single supplier. In this way, the event channel decouples suppliers and consumers.

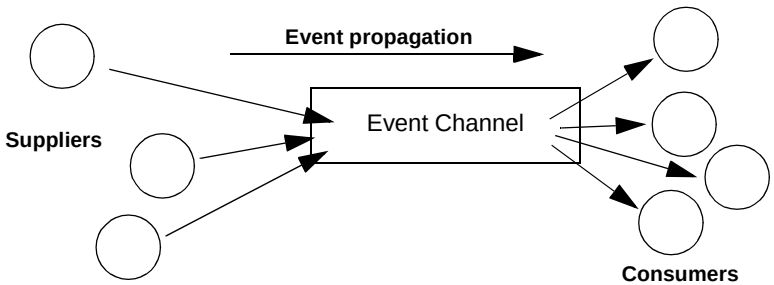


Figure 31: *Suppliers and consumers communicating through an event channel*

How many clients?

Any number of suppliers can issue events to any number of consumers using a single event channel. There is no correlation between the number of suppliers and the number of consumers. New suppliers and consumers can be easily added to or removed from the system. Furthermore, any supplier or consumer can connect to more than one event channel.

For example, many documents might be linked to a spreadsheet cell, and must be notified when the cell value changes. However, the spreadsheet software does not need to know about the documents linked to its cell. When the cell value changes, the spreadsheet software should be able to issue an event that is automatically forwarded to each connected document.

Event delivery

Figure 32 shows a sample implementation of event propagation in a CORBA system. In this example, suppliers are implemented as CORBA clients; the event channel and consumers are implemented as CORBA servers. An event occurs when a supplier invokes a clearly defined IDL operation on an object in the event channel application. The event channel then propagates the event by invoking a similar operation on objects in each of the consumer servers.

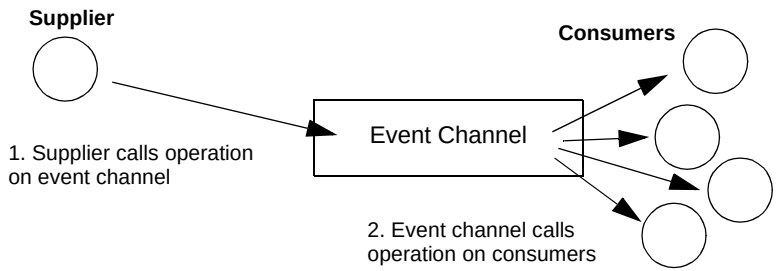


Figure 32: *Event propagation in a CORBA system*

Event Communication Models

Overview

CORBA specifies two approaches to initiating the transfer of events between suppliers and consumers

- Push model: Suppliers initiate transfer of events by sending those events to the channel. The channel then forwards them to any consumers connected to it.
- Pull model: Consumers initiate the transfer of events by requesting them from the channel. The channel requests events from the suppliers connected to it.
- Typed push model: Suppliers initiate the transfer of events by calling operations on an interface that is mutually agreed upon by both the consumer and the supplier. The channel forwards the events to all connected consumers that support the interface.

Push model

In the push model, suppliers generate events and actively pass them to an event channel. In this model, consumers wait for events to arrive from the channel.

Figure 33 illustrates a push model architecture in which push suppliers communicate with push consumers through the event channel.

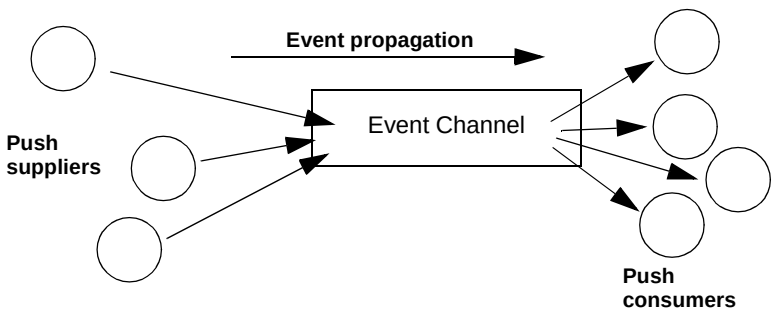


Figure 33: *Push model of event transfer*

In this architecture, a supplier initiates event transfer by invoking an IDL operation on an object in the event channel. The event channel then invokes a similar operation on an object in each consumer that is connected to the channel.

Pull model

In the pull model, a consumer actively requests events from the channel. The supplier waits for a pull request to arrive from the channel. When a pull request arrives, event data is generated and returned to the channel.

Figure 34 illustrates a pull model architecture in which pull consumers communicate with pull suppliers through the event channel.

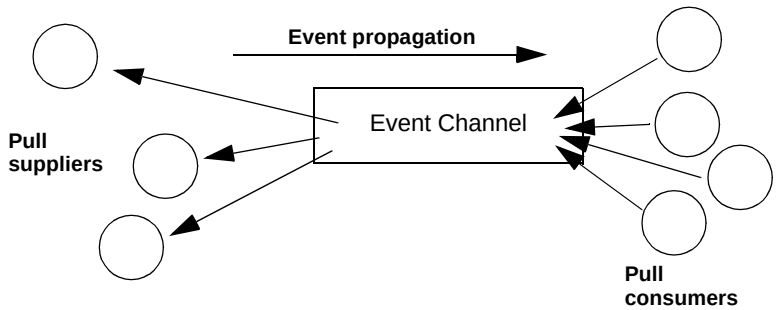


Figure 34: *Pull Model suppliers and consumers communicating through an event channel*

In this architecture, the event channel invokes an IDL operation on an object in each supplier to collect events. When a consumer invokes a similar operation on the event channel, the channel forwards the events to the consumer that initiated the transfer.

Mixing push and pull models

Because suppliers and consumers are completely decoupled by the event channel, push and pull models can be mixed in a single system.

For example, suppliers can connect to an event channel using the push model, while consumers connect using the pull model, as shown in Figure 35.

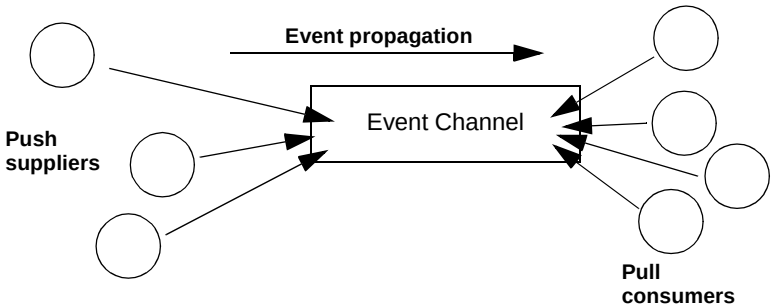


Figure 35: *Push suppliers and pull consumers communicating through an event channel*

In this case, both suppliers and consumers participate in initiating event transfer. A supplier invokes an operation on an object in the event channel to transfer an event to the channel. A consumer then invokes another operation on an event channel object to transfer the event data from the channel.

In the case where push consumers and pull suppliers are mixed, the event channel actively propagates events by invoking IDL operations in objects in both suppliers and consumers. The pull supplier waits for the channel to invoke an event transfer before sending events. Similarly, the push consumer waits for the event channel to invoke event transfer before receiving events.

Typed push model

In the typed push model suppliers connect to the channel using a consumer proxy that supports a user defined interface. The supplier then pushes strongly typed events to the channel by invoking the operations supported by the interface.

Figure 36 shows how typed push suppliers forward events to typed push consumers through a typed event channel. Push suppliers can only forward event messages to typed push consumers that support the agreed upon interface.

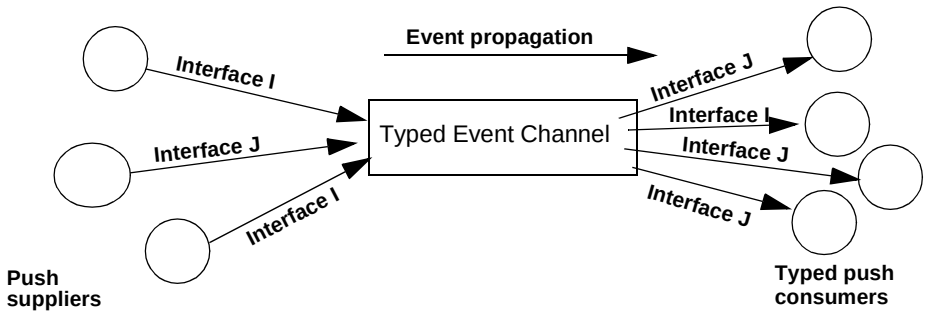


Figure 36: *Push consumers pushing typed events to typed push consumers*

As shown in the diagram, the decoupled nature of the event communication is preserved. Only one typed push consumer supports *Interface I*, but it receives events from two push suppliers. Also, only a single supplier pushes events using *Interface J*, but several typed push consumers support the interface and therefore receive the events.

Developing an Application Using Untyped Events

Overview

When using untyped events messages are packaged into Anys before they are forwarded through the event channel.

In this section

This section discusses the following topics:

Obtaining an Event Channel

Overview

Consumers and suppliers obtain an event channel object reference either by creating a channel, or by finding an existing one.

You obtain an event channel factory by calling `resolve_initial_references("EventChannelFactory")`. You narrow this reference to a event channel factory with Orbix extensions.

Event channel factory

Orbix provides the `EventChannelFactory` interface, which provides the operations to create and discover event channels:

```
module IT_EventChannelAdmin
{
    typedef long ChannelID;

    struct EventChannelInfo
    {
        string                                  name;
        ChannelID                               id;
        CosEventChannelAdmin::EventChannel      reference;
    };
    typedef sequence<EventChannelInfo> EventChannelInfoList;

    exception ChannelAlreadyExists {string name;};
    exception ChannelNotFound {string name;};

  interface EventChannelFactory : IT_MessagingAdmin::Manager
  {
    CosEventChannelAdmin::EventChannel create_channel(
            in  string name,
            out ChannelID id)
     raises (ChannelAlreadyExists);

    CosEventChannelAdmin::EventChannel find_channel(
            in  string  name,
            out ChannelID id)
    raises (ChannelNotFound);
```

```
    CosEventChannelAdmin::EventChannel find_channel_by_id(
            in  ChannelID   id,
            out string      name)
    raises (ChannelNotFound);

    EventChannelInfoList list_channels();
  };
};
```

Event channel factory operations

You can call one of several operations on an event channel factory to create or find an event channel. By providing both create and find operations, the event service allows any client or supplier to create an event channel, which other clients and suppliers can subsequently discover:

create_channel() creates an event channel and returns an object reference.

find_channel() returns an object reference to the named event channel.

find_channel_by_id() returns an object reference to an event channel based on the channel's ID.

list_channels() returns a list of event channels, which provides their names, IDs, and object references.

Example

The following code can be used by any supplier or consumer to obtain an event channel.

Example 59: *Obtaining an event channel*

```
import org.omg.CORBA.*;
import org.omg.CORBA.ORBPackage.*;
import org.omg.CosEventChannelAdmin.*;

//Iona specific classes
import com.iona.corba.IT_EventChannelAdmin.*;

EventChannel ec = null;
EventChannelFactory m_factory = null;
IntHolder id = new IntHolder();
```

Example 59: *Obtaining an event channel*

```
1  Object obj =
     orb.resolve_initial_references("EventChannelFactory");
   m_factory = EventChannelFactoryHelper.narrow(obj);

2  try ec = m_factory.create_named_channel("EventChannel", id)
3  catch (ChannelAlreadyExists cae)
   //Channel already exists, so try to find it
4    try {
         ec = m_factory.find_channel("EventChannel", id);
     }
     catch (ChannelNotFound cnf) {
         System.err.println(
            "Could not create or find event channel");
         System.exit(1);
     }
     catch (SystemException sys){
         System.err.println("System exception occurred during
                         find_channel: " +
                        SystemExceptionDisplayHelper.toString(sys));
         System.exit(1);
     }
```

This code executes as follows:

1. Obtains the event channel factory.
2. Tries to create an event channel by calling `create_named_channel()`.
3. Catches exception `IT_EventChannelAdmin::ChannelAlreadyExists` if a channel of the specified name already exists.
4. Tries to obtain an existing channel of the same name by calling `find_channel()`.

Implementing a Supplier

Actions

A client supplier program performs the following actions:

1. Instantiates suppliers using the appropriate interface in module `CosEventComm`.
2. Connects suppliers to the event channel.
3. Sends event messages to the event channel.
4. Disconnects from the event channel.

Instantiating the Supplier

You instantiate a push supplier with the `PushSupplier` interface; and a pull supplier with the `PullSupplier` interface. Both are defined in the IDL module `CosEventComm`:

Example 60: *Supplier interfaces*

```
module CosEventComm {
   exception Disconnected {};

   interface PullSupplier
   {
      any pull() raises (Disconnected);
      any try_pull (out boolean has_event)
         raises (Disconnected);
      void disconnect_pull_supplier();
   };

   interface PushSupplier
   {
      void disconnect_push_supplier();
   };
};
```

Connecting to a Channel

In order to pass messages to the event channel, a supplier must connect to it through a proxy consumer that receives events from the supplier. Each supplier must have its own proxy consumer. The proxy consumer passes the events down the channel.

A client supplier connects to the event channel in three steps:

1. Obtain a SupplierAdmin object from the event channel.
2. Obtain a proxy consumer in the event channel, to receive the events that the supplier generates.
3. Connect a supplier to a proxy consumer.

Obtain a SupplierAdmin

On creation, an event channel instantiates a default `SupplierAdmin` object, which you obtain by calling `for_suppliers()` on the event channel. For example:

```
org.omg.CosEventChannelAdmin.SupplierAdmin sa =
  channel.for_sppliers();
```

Obtain a proxy consumer

A proxy consumer is responsible for receiving event messages from its client supplier and inserting them into the event channel, where they are forwarded to all interested consumers. You obtain one proxy consumer for each client supplier.

The type of proxy consumer that you obtain depends on whether the client supplier uses the push or pull model. The type of proxy consumer must match the type of its client supplier: a push supplier must use a push proxy consumer; and a pull supplier must use a pull proxy supplier.

The `CosEventChannelAdmin` module supports the two proxy consumer object types with the following interfaces:

```
module CosEventChannelAdmin
{
    exception AlreadyConnected {};
    exception TypeError {};

    interface ProxyPushConsumer : CosEventComm::PushConsumer
    {
        void
        connect_push_supplier(
            in CosEventComm::PushSupplier push_supplier
        ) raises (AlreadyConnected);
    };
```

```
    interface ProxyPullConsumer : CosEventComm::PullConsumer
    {
        void
        connect_pull_supplier(
            in CosEventComm::PullSupplier pull_supplier
        ) raises (AlreadyConnected, TypeError);
    };
    // ...
};
```

You obtain a proxy consumer by invoking one of the following operations on a supplier admin:

obtain_push_consumer() returns a push-model proxy consumer.

obtain_pull_consumer() returns a pull-model proxy consumer.

Example

The following code obtains a `ProxyPushConsumer` for a `PushSupplier` by calling `obtain_push_consumer()`.

Example 61: *Obtaining a proxy consumer*

```
import org.omg.CosEventChannelAdmin.*;

try
{
  ProxyConsumer ppc =
    sa.obtain_push_consumer();
}
```

Connect a supplier to a proxy consumer

After creating a proxy consumer, you can connect it to a compatible client supplier. This establishes the client supplier's connection to the event channel so it can send messages.

Each proxy consumer interface supports a connect operation; the operation requires that the supplier and its proxy support the same delivery model. For example, the `ProxyPushConsumer` interface defines `connect_push_supplier()`, which only accepts an object reference to a `PushSupplier` as input.:

```
interface ProxyPushConsumer : CosEventComm::PushConsumer
{
    void
    connect_push_supplier(
        in CosEventComm::PushSupplier push_supplier
    ) raises (AlreadyConnected);
};
```

Example

The following code shows one way to implement a `PushSupplier` client that connects itself to a proxy consumer.

Example 62: *Connecting a PushSupplier*

```
// proxy ppc and PushSupplier supplier obtained previously
try{
   ppc.connect_push_supplier(supplier);
}
catch (AlreadyConnected.value ac) {
   // Handle the exception
}
catch (SystemException sys){
   System.err.println("Encountered system exception
                  during connect: " +
                  SystemExceptionDisplayHelper.toString(sys));
   System.exit(1);
}
```

Sending Event Messages

A client supplier sends event messages in one of two ways:

- A push supplier invokes the `push` operation on its proxy consumer and supplies the event as an input argument.
- A pull supplier implements `try_pull()`. When the proxy consumer invokes a pull operation, the supplier returns an event message if one is available.

Push supplier

A push supplier invokes the `push()` operation on its proxy consumer. For example:

Example 63: *Pushing an event message*

```
// proxy consumer and event message already obtained
try{
   proxy.push(event_msg);
}
catch (SystemException sys){
   System.err.println("Unexpected system exception during push:"
                      +SystemExceptionDisplayHelper.toString(sys));
   System.exit(1);
}
catch (org.omg.CosEventComm.Disconnected dc){
   System.err.println("Channel is disconnected.");
   System.exit(1);
}
catch (Exception e){
   System.err.println("Unknown exception occurred during push");
   System.exit(1);
}
```

Pull supplier

A pull supplier sends event messages only on request. Whether a client consumer invokes `pull()` or `try_pull()`, the pull supplier's proxy consumer always invokes `try_pull()` on its supplier.

Pull suppliers are responsible for implementing `try_pull()`, which returns a `CORBA::Any`. This operation is non-blocking; it returns immediately with an output parameter of type boolean to indicate whether the return value actually contains an event.

For example, the following code implements `try_pull()` by attempting to populate an event message with the latest baseball scores.

Example 64: *Pulling events*

```
class PullSupplier extends PullSupplierPOA
{
// ...
   public Any try_pull(
      BooleanHolder has_event)
   {
      has_event.value = false;

     // get scores
     String scores;
     boolean has_scores = get_scores(scores);

     // If there are scores, send event message
     if (has_scores == true)
     {
        CORBA.Any event_msg = ORB.create_any();
        event_msg.insert_string(scores);
        has_event.value = true;
     }
     return event_msg;
  }
```

Disconnecting From the Event Channel

A client supplier can disconnect from the event channel at any time by invoking the disconnect operation on its proxy consumer. This operation terminates the connection between a supplier and its target proxy consumer. The channel then releases all resources allocated to support its connection to the supplier, including destruction of the target proxy consumer.

Each proxy consumer interface supports a disconnect operation. For example, interface `ProxyPushConsumer` defines `disconnect_push_consumer()`.

Implementing a Consumer

Actions

A client consumer program performs the following actions:

1. Instantiates consumers with the appropriate `CosEventComm` interface.
2. Connects consumers to the event channel.
3. Obtains event messages.
4. Disconnects from the event channel.

Instantiating a Consumer

You instantiate a push consumer with the `PushConsumer` interface; and a pull consumer with the `PullConsumer` interface. Both are defined in the IDL module CosEventComm:

Example 65: *Consumer interfaces*

```
module CosEventComm
{
    exception Disconnected { };

    interface PushConsumer {
        void push( in any data) raises (Disconnected);

        void disconnect_push_consumer ();
    };

    interface PullConsumer {
        void disconnect_pull_consumer();

    };
};
```

Connecting to the Channel

Consumers receive messages from the event channel through a proxy supplier. Each consumer on the channel has its own proxy supplier. Proxy suppliers use the same delivery method as their consumers and send the appropriate message type.

Consumers connect to the event channel in three steps:

1. Obtain a ConsumerAdmin object from the event channel.
2. Obtain a proxy supplier in the event channel, to receive supplier-generated event messages.
3. Connect the consumer to a proxy supplier.

Obtain a ConsumerAdmin

On creation, an event channel instantiates a default `ConsumerAdmin` object, which you obtain by calling `for_consumers()` on the event channel. For example:

```
org.omg.CosEventChannelAdmin.ConsumerAdmin ca =
  channel.for_consumers();
```

Obtain a proxy supplier

A proxy supplier is responsible for distributing event messages that have been sent by the event channel to its consumer. You create one proxy supplier for each client consumer.

The type of proxy supplier that you obtain depends on whether the client consumer uses the push or pull model. The type of proxy supplier must match the type of its client consumer: a push consumer must use a push proxy supplier; and a pull consumer must use a pull proxy supplier.

The `CosEventChannelAdmin` module supports the two proxy supplier object types with the following interfaces:

Example 66: *Proxy supplier interfaces*

```
module CosEventChannelAdmin
{
    exception AlreadyConnected {};
    exception TypeError {};

    interface ProxyPullSupplier : CosEventComm::PullSupplier
    {
        void
        connect_pull_consumer(
            in CosEventComm::PullConsumer pull_consumer
        ) raises (AlreadyConnected);
    };
```

Example 66: *Proxy supplier interfaces*

```
    interface ProxyPushSupplier : CosEventComm::PushSupplier
    {
        void
        connect_push_consumer(
            in CosEventComm::PushConsumer push_consumer
        ) raises (AlreadyConnected, TypeError);
    };
};
```

You obtain a proxy supplier by invoking one of the following operations on a consumer admin:

obtain_push_supplier() returns a push-model proxy supplier.

obtain_pull_supplier() returns a pull-model proxy supplier.

Example

The following code obtains a proxy supplier for a `PushConsumer` by calling `obtain_push_supplier()`.

Example 67: *Obtaining a proxy supplier*

```
import org.omg.CosEventChannelAdmin.*;
try
{
  ProxySupplier pps =
    ca.obtain_push_supplier();
}
```

Connect the consumer to a proxy supplier

After creating a proxy supplier, you can connect it to a compatible client consumer. This establishes the client's connection to the event channel, so it can obtain messages from suppliers.

Each proxy supplier interface supports a connect operation; the operation requires that the client supplier and its proxy support the same push or pull model and event-message type. For example, the `ProxyPushSupplier` interface defines `connect_push_consumer()`, which only accepts an object reference to a `PushConsumer` as input:

```
interface ProxyPushSupplier :
    ProxySupplier,
    CosEventComm::PushSupplier
{
  void connect_push_consumer
       (in CosEventComm::PushConsumer push_consumer)
  raises(CosEventChannelAdmin::AlreadyConnected,
         CosEventChannelAdmin::TypeError);
};
```

Example

The following example shows how you might implement a `PushConsumer` client that connects itself to a proxy supplier.

Example 68: *Connecting to a proxy supplier*

```
import org.omg.CosEventChannelAdmin.*;

class PushConsumer extends PushConsumerPOA
{
   // ...

   public static void main (String args[])
   {
      // ...
      //Proxy pps and PushConsumer consumer obtained previously
      try{
         pps.connect_push_consumer(consumer);
      }
      catch (AlreadyConnected.value ac){
         System.err.println("Already connecting to channel.");
         System.exit (1);
      }
```

Example 68: *Connecting to a proxy supplier*

```
      catch (SystemException sys){
         System.err.println(
            "Encountered system exception during connect: "
            + SystemExceptionDisplayHelper.toString(sys));
         System.exit(1);
      }
      //...
   }
}
```

Obtaining Event Messages

A client consumer obtains event messages in one of two ways:

- A push consumer implements the `push()` operation. As events become available, the proxy supplier pushes them to its client consumer.
- A pull consumer invokes `pull()` or `try_pull()` on its proxy supplier; the proxy supplier returns with the next available event.

Push consumer

A push consumer implements the `push()` operation. For example:

Example 69: *Receiving events using push()*

```
class PushConsumer extends PushConsumerPOA
{
   // ...
   public void push(Any  event)
   {
      String scores = event.extract_string();
      System.out.println("Current " + sports_type + "scores:
                              " + scores);
   }
   //...
}
```

Pull consumer

A pull client consumer invokes the `pull()` or `try_pull()` operation on its proxy supplier to solicit event messages; the proxy supplier returns with the next available event.

The proxy supplier interface supports operations `pull()` and `try_pull()`. A pull consumer invokes one of these operations on its `ProxyPullSupplier`. Both operations return a `CORBA::Any` argument; they differ only in their blocking mode:

pull() blocks until an event is available.

try_pull() is non-blocking—it returns immediately with a boolean output parameter to indicate whether the return value actually contains an event. The event channel continues to invoke the pull operation on suppliers until one of them supplies an event. When an event becomes available, `try_pull()` sets its boolean `has_event` parameter to true and returns with the event data to the pull consumer.

The following example shows how a pull consumer might invoke `try_pull()` to receive data from its `ProxyPullSupplier`.

Example 70: *Pulling events*

```
Any scores = null;
BooleanHolder has_data = new BooleanHolder();

try{
    event = proxy.try_pull(has_data);
}
catch (org.omg.CosEventComm.Disconnected dsc ){
   System.err.println("Disconnected exception occured during
                       pull");
   System.exit (1);
}
catch (SystemException sys ){
   System.err.println("System exception occured during pull");
   System.exit (1);
}

if (has_data.value)
{
   scores = event.extract_string();
   System.out.println("Received event number " + scores
           + " using try pull");
}
```

Disconnecting From the Event Channel

A client consumer can disconnect from the event channel at any time by invoking the disconnect operation on its proxy supplier. This operation terminates the connection between the consumer and its target proxy supplier. The event channel then releases all resources allocated to support its connection to the consumer, including destruction of the target proxy supplier.

Each proxy supplier interface supports a disconnect operation. For example, interface `ProxyPushSupplier` defines `disconnect_push_supplier()`.

Developing an Application Using Typed Events

Overview

Typed events allow event service clients to use a strongly typed interface to pass events back and forth. Using typed events can increase the performance of event service clients by eliminating the time used for marshalling, encoding, unmarshalling, and decoding of events packaged into Anys. Typed event clients can also use non-typed event communication to send and receive messages.

In this section

This section discusses the following topics:

Creating the Interface

Overview

When using typed push event communication, suppliers and consumers use a mutually agreed upon interface to facilitate event forwarding. This interface is defined in IDL and stored in the interface repository.

Interface restrictions

Because typed event communication is strictly from the supplier to the consumer, there are two restrictions on the operations of an interface used for typed event communication:

- They can only have `in` parameters.
- They cannot have a return type other than `void`.

Messages cannot be passed through the event channel from consumer to supplier and these restrictions help reinforce the unidirectional nature of event forwarding.

Example

The interface, `ScorePusher`, in Example 71 shows a simple interface to push a sports score.

Example 71: *Typed event interface ScorePusher*

```
\\IDL
interface ScorePusher
{
  void push_score(in string team_a, in long score_a,
                  in string team_b, in long score_b);
};
```

Once you have written the interface, you must place it into the interface repository using the following command:

```
idl -R filename
```

Obtaining a Typed Event Channel

Overview

A typed event channel forwards messages between typed event clients. It provides the same operations as the untyped event channel.

Consumers and suppliers obtain a typed event channel object reference either by creating a channel, or by finding an existing one.

You obtain a typed event channel factory by calling `resolve_initial_references("EventChannelFactory")`. You narrow the returned reference to a typed event channel factory with Orbix extensions.

Event channel factory

Orbix provides the `TypedEventChannelFactory` interface, which define the operations to create and discover typed event channels:

```
module IT_TypedEventChannelAdmin
{
  struct TypedEventChannelInfo
  {
    string                                        name;
    IT_EventChannelAdmin::ChannelID               id;
    CosTypedEventChannelAdmin::TypedEventChannel reference;
  };
  typedef sequence<TypedEventChannelInfo>
   TypedEventChannelInfoList;

  interface TypedEventChannelFactory :
            IT_MessagingAdmin::Manager
  {
    CosTypedEventChannelAdmin::TypedEventChannel
    create_typed_channel(in  string name,
                         out IT_EventChannelAdmin::ChannelID id)
    raises(IT_EventChannelAdmin::ChannelAlreadyExists);

    CosTypedEventChannelAdmin::TypedEventChannel
    find_typed_channel(in  string name,
                         out IT_EventChannelAdmin::ChannelID id)
    raises(IT_EventChannelAdmin::ChannelNotFound);
```

```
    CosTypedEventChannelAdmin::TypedEventChannel
    find_typed_channel_by_id(
      in  IT_EventChannelAdmin::ChannelID id,
      out string                          name)
    raises(IT_EventChannelAdmin::ChannelNotFound);

    TypedEventChannelInfoList list_typed_channels();
  };
};
```

Typed event channel factory operations

You can call one of several operations on an event channel factory to create or find an event channel. By providing both create and find operations, the event service allows any client or supplier to create an event channel, which other clients and suppliers can subsequently discover:

create_typed_channel() creates a typed event channel and returns an object reference.

find_typed_channel() returns an object reference to the named typed event channel.

find_typed_channel_by_id() returns an object reference to a typed event channel based on the channel's ID.

list_typed_channels() returns a list of typed event channels, which provides their names, IDs, and object references.

Example

The following code can be used by any supplier or consumer to obtain a typed event channel.

Example 72: *Obtaining a typed event channel*

```
import org.omg.CORBA.*;
import org.omg.CORBA.ORBPackage.*;
import org.omg.CosTypedEventChannelAdmin.*;
```

Example 72: *Obtaining a typed event channel*

```
//Iona specific classes
import org.omg.CosEventChannelAdmin.*;
import com.iona.corba.IT_EventChannelAdmin.*;
import com.iona.corba.IT_TypedEventChannel.*;

TypedEventChannel tec = null;
TypedEventChannelFactory m_factory = null;
IntHolder id = new IntHolder();

1 try
{
  Object obj = orb.resolve_initial_references("EventService");
}
catch (InvalidName)
{
 // Handle the exception
}
m_factory = TypedEventChannelFactoryHelper.narrow(obj);

2 try
{
  tec = m_factory.create_typed_channel("TypedChannel", id);
}
3 catch (ChannelAlreadyExists cae)
{
//Channel already exists, so try to find it
4   try
  {
    tec = m_factory.find_typed_channel("TypedChannel", id);
  {
  catch (ChannelNotFound cnf)
  {
    System.err.println("Could not create or find event channel");
    System.exit(1);
  }
  catch (SystemException sys)
  {
    System.err.println("System exception occurred during
                        find_channel: " +
                      SystemExceptionDisplayHelper.toString(sys));
    System.exit(1);
  }
}
```

This code executes as follows:

1. Obtains the typed event channel factory.
2. Tries to create a typed event channel by calling `create_typed_channel()`.
3. Catches exception `IT_EventChannelAdmin::ChannelAlreadyExists` if a channel of the specified name already exists.
4. Tries to obtain an existing channel of the same name by calling `find_typed_channel()`.

Implementing the Supplier

Actions

The actions performed by a push supplier for typed event communications are similar to the actions performed by a push supplier for untyped event communication. These actions are:

1. Instantiate an instance of the `CosEventComm::PushSupplier` interface.
2. Connect to a typed event channel.
3. Push typed event messages by obtaining the appropriate interfaces and invoking its operations.
4. Disconnect from the typed event channel.

Instantiate the supplier

Typed push style event communication uses a generic push supplier to supply events to typed push consumers. An application that is intended to push typed events to typed event consumers can instantiate an instance of the `CosEventComm::PushSupplier` interface.

If the supplier does not need to be informed if its proxy disconnects from the channel, the supplier can connect a `null` to the typed proxy consumer.

Connecting to a typed event channel

In order to pass messages to the typed event channel, a supplier must connect to it through a typed proxy consumer that receives events from the supplier. The proxy consumer passes the events down the channel.

A supplier connects to the typed event channel in three steps:

1. Obtain a TypedSupplierAdmin from the typed event channel.
2. Obtain a typed proxy consumer in the typed event channel, to receive the events generated by the supplier.
3. Connect a supplier to a typed proxy consumer.

Obtain a TypedSupplierAdmin

On creation, a typed event channel instantiates a default `TypedSupplierAdmin`, which you obtain by calling `for_suppliers()` on the typed event channel. For example:

```
org.omg.CosTypedEventChannelAdmin.TypedSupplierAdmin tsa =
  tec.for_suppliers();
```

Obtain a typed proxy consumer

A typed proxy consumer is responsible for receiving typed event messages from its supplier and inserting them into the event channel, where they are forwarded to all interested typed consumers. You obtain one typed proxy consumer for each client supplier.

The `CosTypedEventChannelAdmin` module supports the typed proxy push consumer object type with the following interfaces:

```
module CosTypedEventChannelAdmin
{
  exception InterfaceNotSupported {};
  exception NoSuchImplementation {};

  interface TypedProxyPushConsumer :
            CosTypedEventComm::TypedPushConsumer,
            CosEventChannelAdmin::ProxyPushConsumer
  {
  };
}
```

You obtain a typed proxy consumer by invoking `obtain_typed_push_consumer()` on a typed supplier admin and supplying the interface repository ID of the interface the supplier intends to use to push events. If there are no consumers on the typed event channel which support the specified interface a `InterfaceNotSupported` exception is raised.

Example

The following code obtains a `TypedProxyPushConsumer` for a `PushSupplier` by calling `obtain_typed_push_consumer()`.

Example 73: *Obtaining a proxy consumer*

```
import org.omg.CosTypedEventChannelAdmin.*;

try
{
  TypedProxyConsumer tpc =
   tsa.obtain_typed_push_consumer("IDL:ScorePusher:1.0");
}
catch (InterfaceNotSupported)
{
  // handle the exception
}
```

Connect a supplier to a typed proxy consumer

After creating a typed proxy consumer, you can connect it to a compatible supplier. This establishes the supplier's connection to the typed event channel so it can send messages.

Typed proxy consumers support the `connect_push_supplier()` operation. The operation requires that the supplier and its proxy support the same interface.

Example 74 shows one way to implement a `PushSupplier` client that connects itself to a typed proxy consumer.

Example 74: *Connecting a PushSupplier*

```
// proxy tpc and PushSupplier supplier obtained previously
try{
   tpc.connect_push_supplier(supplier);
}
catch (AlreadyConnected ac) {
   // Handle the exception
}
catch (SystemException sys){
   System.err.println("Encountered system exception
                during connect: " +
                SystemExceptionDisplayHelper.toString(sys));
   System.exit(1);
}
```

Pushing typed events

In typed push event communication the supplier pushes events to the consumers by invoking operations on an interface that has been mutually agreed upon by both the developer responsible for implementing the supplier and the developer responsible for implementing the consumer.

The supplier obtains a reference to the appropriate interface by invoking its associated typed proxy consumer's `get_typed_consumer()` operation. This operation returns a reference to the interface specified when `obtain_typed_push_consumer()` was invoked to obtain the typed proxy consumer. The returned reference is of type `Object` and must be narrowed to the appropriate interface.

Note: If the supplier and the client do not support the identical interface the `narrow()` operation will fail.

Example 75 shows how a push supplier would pass typed messages to typed consumers that supported the `ScorePusher` interface defined earlier.The above code performs the following actions:

Example 75: *Pushing typed events using the ScorePusher interface*

```
// Java
import org.omg.CORBA.*;
import org.omg.CosTypedEventComm.*;
import org.omg.CosTypedEventChannelAdmin.*;
1 Object obj = tpc.get_typed_consumer();
2 ScorePusher pusher = ScorePusherHelper.narrow(obj);
3 pusher.push_score("Hooligans", 9, "Ruffians", 12);
```

1. Obtains a reference to an appropriate typed consumer interface.
2. Narrows the reference.
3. Invokes the `push_score()` operation to forward the event to any typed push consumers that implement the `ScorePusher` interface.

Disconnecting From the Event Channel

A supplier can disconnect from a typed event channel at any time by invoking the `disconnect_push_consumer()` operation. This operation terminates the connection between a supplier and its target typed proxy consumer. The channel then releases all resources allocated to support its connection to the supplier and destroys the target typed proxy consumer.

Implementing the Consumer

Overview

In typed push style event communication the consumer is responsible for implementing the interface that is used to forward events. Also, the consumer is instantiated using a typed event interface, `CosTypedEventComm::TypedPushConsumer`, instead of the generic push consumer interface.

Development tasks

The developer of a typed push consumer must complete the following tasks:

- Implement the mutually agreed upon interface.
- Instantiate the consumer using the `CosTypedEventComm::TypedPushConsumer` interface.
- Connect the consumer to a typed event channel.
- Receive event messages from the channel and process them.
- Disconnect the consumer from the typed event channel.

Implement the interface

The first step in developing a typed push consumer is to implement the interface that will be used to support the typed events. To do this complete the following steps:

1. Create a new IDL interface that inherits from the interface that will be used for event communication and from `CosEventComm::PushConsumer`. For the `ScorePusher` interface the combined interface for the consumer might look like:

```
\\IDL
#include <ScorePusher.idl>
#include <omg/CosEventComm.idl>

interface ScoreConsumer : ScorePusher,
                          CosEventComm::PushConsumer
{
};
```

2. Compile the IDL interface into the desired programming language.
3. Implement the operation to be used for forwarding typed events.

4. Implement `push()`. If the consumer participate exclusively in typed event communication, `push()` can do nothing.

For example, the code shown in Example 76 shows one way to implement a typed push consumer that uses the `ScorePusher` interface to forward events.

Example 76: *Implementing a typed push consumer*

```
// Java
import org.omg.CORBA.Orb.*;
import org.omg.CosTypedEventChannelAdmin.*;
import com.iona.IT_TypedEventChannelAdmin.*;
class ScoreConsumer extends ScoreConsmuerPOA
{
  // constructor and destructor
  // ...

3   void push_score(String team_a, int score_a,
                    String team_b, int score_b)
  {
    System.out.println("Score:");
    System.out.println(team_a + "\t" + score_a);
    System.out.println(team_b + "\t" + score_b);
  }

4   void push(org.omg.CORBA.Any a)
  {
  }

  void disconnect_push_consumer()
  {
  }
  // implement the main()
  // ...
}
```

Instantiate the consumer

Typed push event communication uses the `CosTypedEventComm::TypedPushConsumer` interface to receive events. Clients wishing to act as consumers in typed push style events must instantiate an instance of this interface or, as above, an interface that inherits from it. Using the example above, the application would instantiate an instance of `ScoreConsumer` which implements both the interface used to forward events and `CosTypedEventComm::TypedPushConsumer`.

Connecting to the channel

Typed push consumers connect to a typed event channel through a proxy push supplier which receives the events from the channel and forwards them to the consumer.

The steps to connect a typed push consumer to a typed event channel are the same as the steps to connect a generic consumer to an event channel, They are:

1. Obtain a typed consumer admin object from the typed event channel.
2. Obtain a proxy push supplier from the consumer admin.
3. Connect the consumer to the proxy supplier.

Obtain a typed consumer admin

On creation, a typed event channel instantiates a default `TypedConsumerAdmin` object, which you obtain by calling `for_consumers()` on the event channel. For example:

```
org.omg.CosTypedEventChannelAdmin.TypedConsumerAdmin tca =
  tec.for_consumers();
```

Obtain a proxy supplier

A proxy push supplier is responsible for distributing event messages that have been sent by the typed event channel to its typed consumer. You create one proxy supplier for each client consumer.

You obtain a proxy push supplier by invoking `obtain_typed_push_supplier()` on the typed consumer admin and supplying the interface's interface repository id. For example, to obtain a proxy push supplier for use with the `ScorePusher` interface, you would use the following operation:

```
try
{
  CosEventChannelAdmin::ProxyPushSupplier pps =
   tca->obtain_typed_push_supplier("IDL:ScorePusher:1.0");
}
catch (CosTypedEventChannelAdmin::NoSuchImplementation)
{
  // no push supplier implements the appropriate interface
  // handle the exception
}
```

```
try
{
  org.omg.CosEventChannelAdmin.ProxyPushSupplier pps =
   tca.obtain_typed_push_supplier("IDL:ScorePusher:1.0");
}
catch (CosTypedEventChannelAdmin.NoSuchImplementation)
{
  // no supplier implements the interface
  // handle the exception
}
```

Connect the consumer to a proxy supplier

After creating a proxy push supplier, you can connect it to a client consumer. This establishes the client's connection to the typed event channel, so it can obtain messages from suppliers.

The proxy push supplier interface supports the connect operation `connect_push_consumer()`, which accepts an object reference to a `TypedPushConsumer` as input.

Example 77 shows how you might implement a `TypedPushConsumer` client that connects itself to a proxy supplier.

Example 77: *Connecting to a proxy supplier*

```
import org.omg.CosEventChannelAdmin.*;

class PushConsumer extends PushConsumerPOA
{
   // ...

   public static void main (String args[])
   {
      // ...
      //Proxy pps and PushConsumer consumer obtained previously
      try{
         pps.connect_push_consumer(consumer);
      }
      catch (AlreadyConnected.value ac){
         System.err.println("Already connecting to channel.");
         System.exit (1);
      }
```

Example 77: *Connecting to a proxy supplier*

```
      catch (org.omg.CosEventChannelAdmin.TypeError)
      {
         System.err.println(
            "Encountered system exception during connect: "
            + SystemExceptionDisplayHelper.toString(sys));
         System.exit(1);
      }
      //...
   }
}
```

Receiving event messages

Typed push consumers passively receive messages from the channel. As events become available the proxy supplier forwards them to the consumer using one of the operations in the mutually agreed upon interface. The operation, which was implemented previously, is responsible for processing the event.

Disconnecting from the event channel

A client consumer can disconnect from the event channel at any time by invoking `disconnect_push_consumer()`. This operation terminates the connection between the consumer and its target proxy supplier. The typed event channel then releases all resources allocated to support its connection to the consumer and destroys the target proxy supplier.

Portable Interceptors

Portable interceptors provide hooks, or interception points, which define stages within the request and reply sequence. Services can use these interception points to query request/reply data, and to transfer service contexts between clients and servers.

Sample application

This chapter shows an application that uses interceptors to secure a server with a password authorization service as follows:

- A password policy is created and set on the server's POA.
- An IOR interceptor adds a *tagged component* to all object references exported from that POA. This tagged component encodes data that indicates whether a password is required.
- A client interceptor checks the profile of each object reference that the client invokes on. It ascertains whether the object is password-protected; if so, it adds to the outgoing request a service context that contains the password data.

- A server interceptor checks the service contexts of incoming requests for password data, and compares it with the server password. The interceptor allows requests to continue only if the client and server passwords match.

Note: The password authorization service that is shown here is deliberately simplistic, and intended for illustrative purposes only.

In this chapter

This chapter contains the following sections:

Interceptor Components

Portable interceptors require the following components:

Interceptor implementations that are derived from interface `PortableInterceptor::Interceptor`.

IOP::ServiceContext supplies the service context data that a client or server needs to identify and access an ORB service.

PortableInterceptor::Current (hereafter referred to as *PICurrent*) is a table of slots that are available to application threads and interceptors, to store and access service context data.

IOP::TaggedComponent contains information about optional features and ORB services that an IOR interceptor can add to an outgoing object reference. This information is added by server-side IOR interceptors, and is accessible to client interceptors.

IOP::Codec can convert data into an octet sequence, so it can be encoded as a service context or tagged component.

PortableInterceptor::PolicyFactory enables creation of policy objects that are required by ORB services.

PortableInterceptor::ORBInitializer is called on ORB initialization. An ORB initializer obtains the ORB's PICurrent, and registers portable interceptors with the ORB. It can also register policy factories.

Interceptor Types

All portable interceptors are based on the `Interceptor` interface:

```
module PortableInterceptor{
    local interface Interceptor{
        readonly attribute string name;
    };
};
```

An interceptor can be named or unnamed. Among an ORB's interceptors of the same type, all names must be unique. Any number of unnamed, or anonymous interceptors can be registered with an ORB.

Note: At present, Orbix provides no mechanism for administering portable interceptors by name.

All interceptors implement one of the interceptor types that inherit from the `Interceptor` interface:

ClientRequestInterceptor defines the interception points that client-side interceptors can implement.

ServerRequestInterceptor defines the interception points that server-side interceptors can implement.

IORInterceptor defines a single interception point, `establish_components`. It is called immediately after a POA is created, and pre-assembles the list of tagged components to add to that POA's object references.

Interception points

Each interceptor type defines a set of interception points, which represent stages in the request/reply sequence. Interception points are specific to each interceptor type, and are discussed fully in later sections that describe these types. Generally, in a successful request-reply sequence, the ORB calls interception points on each interceptor.

For example, Figure 37 shows client-side interceptors A and B. Each interceptor implements interception points `send_request` and `receive_reply`. As each outgoing request passes through interceptors A and B, their `send_request` implementations add service context data `a` and `b` to

the request before it is transported to the server. The same interceptors' `receive_reply` implementations evaluate the reply's service context data before the reply returns to the client.

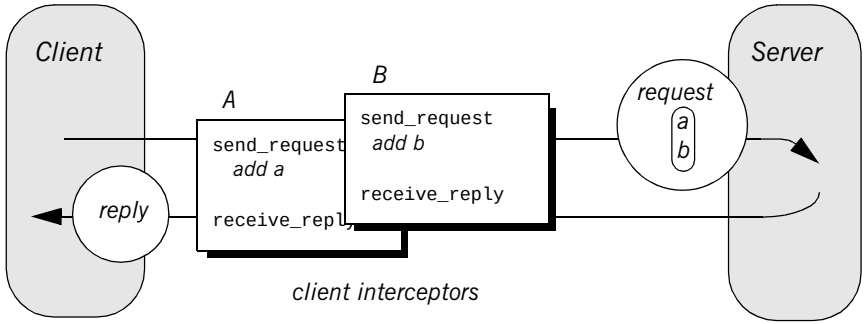


Figure 37: *Client interceptors allow services to access outgoing requests and incoming replies.*

Interception point data

For each interception point, the ORB supplies an object that enables the interceptor to evaluate the request or reply data at its current stage of flow:

- A `PortableInterceptor::IORInfo` object is supplied to an IOR interceptor's single interception point `establish_components` (see page 528).
- A `PortableInterceptor::ClientRequestInfo` object is supplied to all `ClientRequestInterceptor` interception points (see page 542).
- A `PortableInterceptor::ServerRequestInfo` object is supplied to all `ServerRequestInterceptor` interception points (see page 551).

Much of the information that client and server interceptors require is similar; so `ClientRequestInfo` and `ServerRequestInfo` both inherit from interface `PortableInterceptor::RequestInfo`. For more information on `RequestInfo`, see page 532.

Service Contexts

Service contexts supply the information a client or server needs to identify and access an ORB service. The IOP module defines the `ServiceContext` structure as follows:

Example 78: *ServiceContext structure*

```
module IOP
{
    // ...
    typedef unsigned long ServiceId;

    struct ServiceContext {
        ServiceId context_id;
        sequence <octet> context_data;
    };
};
```

A service context has two member components:

- Service-context IDs are user-defined unsigned long types. The high-order 20 bits of a service-context ID contain a 20-bit vendor service context codeset ID, or *VSCID*; the low-order 12 bits contain the rest of the service context ID. To define a set of service context IDs:
  i. Obtain a unique VSCID from the OMG
  ii. Define the service context IDs, using the VSCID for the high-order bits.
- Service context data is encoded and decoded by an `IOP::Codec` (see "Codec" on page 524).

PICurrent

PICurrent is a table of slots that different services can use to transfer their data to request or reply service contexts. For example, in order to send a request to a password-protected server, a client application can set the required password in PICurrent. On each client invocation, a client interceptor's `send_request` interception point obtains the password from PICurrent and attaches it as service context data to the request.

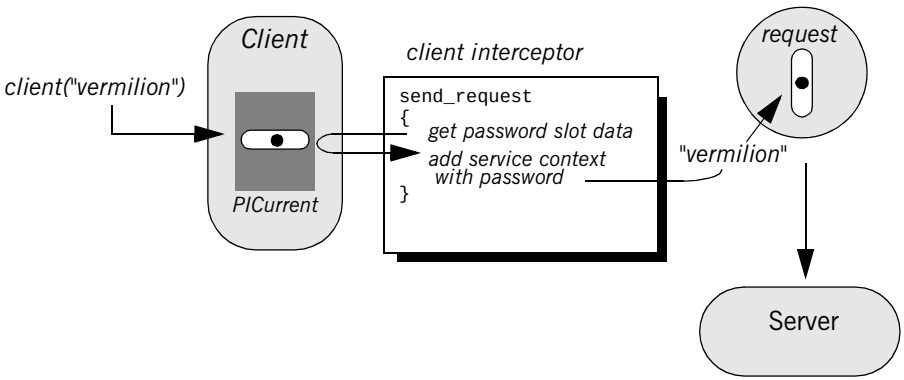


Figure 38: *PICurrent facilitates transfer of thread context data to a request or reply.*

Interface definition

The `PortableInterceptor` module defines the interface for PICurrent as follows:

Example 79: *PortableInterceptor:Current (PICurrent) interface*

```
module PortableInterceptor
{
    // ...
    typedef unsigned long SlotId;
    exception InvalidSlot {};
```

Example 79: *PortableInterceptor:Current (PICurrent) interface*

```
    local interface Current : CORBA::Current {
        any
        get_slot(in SlotId id
        ) raises (InvalidSlot);

        void
        set_slot(in SlotId id, in any    data
        ) raises (InvalidSlot);
    };
};
```

Tagged Components

Object references that support an interoperability protocol such as IIOP or SIOP can include one or more tagged components, which supply information about optional IIOP features and ORB services. A tagged component contains an identifier, or *tag*, and component data, defined as follows:

Example 80: *TaggedComponent structure*

```
typedef unsigned long ComponentId;
struct TaggedComponent{
    ComponentID tag;
    sequence<octet> component_data;
};
```

An IOR interceptor can define tagged components and add these to an object reference's profile by calling `add_ior_component()` (see "Writing IOR Interceptors" on page 528). A client interceptor can evaluate tagged components in a request's object reference by calling `get_effective_component()` or `get_effective_components()` (see "Evaluating tagged components" on page 545).

Note: The OMG is responsible for allocating and registering the tag IDs of tagged components. Requests to allocate tag IDs can be sent to tag_request@omg.org.

Codec

Interface definition

The data of service contexts and tagged components must be encoded as a CDR encapsulation. Therefore, the IOP module defines the Codec interface, so interceptors can encode and decode octet sequences:

Example 81: *Codec interface*

```
local interface Codec {
    exception InvalidTypeForEncoding {};
    exception FormatMismatch {};
    exception TypeMismatch {};

    CORBA::OctetSeq
    encode(in any data
    ) raises (InvalidTypeForEncoding);

    any
    decode(in CORBA::OctetSeq data
    ) raises (FormatMismatch);

    CORBA::OctetSeq
    encode_value(in any data
    ) raises (InvalidTypeForEncoding);

    any
        decode_value(
        in CORBA::OctetSeq data,
        in CORBA::TypeCode tc
    ) raises (FormatMismatch, TypeMismatch);
};
```

Codec operations

The Codec interface defines the following operations:

encode converts the supplied any into an octet sequence, based on the encoding format effective for this Codec. The returned octet sequence contains both the TypeCode and the data of the type.

decode decodes the given octet sequence into an any, based on the encoding format effective for this Codec.

encode_value converts the given any into an octet sequence, based on the encoding format effective for this Codec. Only the data from the any is encoded.

decode_value decodes the given octet sequence into an any based on the given TypeCode and the encoding format effective for this Codec.

Creating a codec

The `ORBInitInfo::codec_factory` attribute returns a Codec factory, so you can provide Codec objects to interceptors. This operation must be called during ORB initialization, through the ORB initializer.

Policy Factory

An ORB service can be associated with a user-defined policy. The `PortableInterceptor` module provides the `PolicyFactory` interface, which applications can use to implement their own policy factories:

```
local interface PolicyFactory {
    CORBA::Policy
    create_policy(
        in CORBA::PolicyType type,
        in any               value
    ) raises (CORBA::PolicyError);
};
```

Policy factories are created during ORB initialization, and registered through the ORB initializer (see "Create and register policy factories" on page 566).

ORB Initializer

ORB initializers implement interface `PortableInterceptor::OrbInitializer`:

Example 82: *ORBInitializer interface*

```
local interface ORBInitializer {
    void
    pre_init(in ORBInitInfo info);

    void
    post_init(in ORBInitInfo info);
};
```

As it initializes, the ORB calls the ORB initializer's `pre_init()` and `post_init()` operations. `pre_init()` and `post_init()` both receive an `ORBInitInfo` argument, which enables implementations to perform these tasks:

- Instantiate a PICurrent and allocates its slots for service data.
- Register policy factories for specified policy types.
- Create `Codec` objects, which enable interceptors to encode service context data as octet sequences, and vice versa.
- Register interceptors with the ORB.

Writing IOR Interceptors

IOR interceptors give an application the opportunity to evaluate a server's effective policies, and modify an object reference's profiles before the server exports it. For example, if a server is secured by a password policy, the object references that it exports should contain information that signals to potential clients that they must supply a password along with requests on those objects.

The IDL interface for IOR interceptors is defined as follows:

```
local interface IORInterceptor : Interceptor {
    void
    establish_components(in IORInfo info);
};
```

Interception point

An IOR interceptor has a single interception point, `establish_components()`. The server-side ORB calls `establish_components()` once for each POA on all registered IOR interceptors. A typical implementation of `establish_components()` assembles the list of components to include in the profile of all object references that a POA exports.

An implementation of `establish_components()` must not throw exceptions. If it does, the ORB ignores the exception.

IORInfo

`establish_components()` gets an `IORInfo` object, which has the following interface:

Example 83: *IORInfo interface*

```
local interface IORInfo {

    CORBA::Policy
    get_effective_policy(in CORBA::PolicyType type);

    void
    add_ior_component(in IOP::TaggedComponent component);
```

Example 83: *IORInfo interface*

```
    add_ior_component_to_profile (
        in IOP::TaggedComponent component,
        in IOP::ProfileId       profile_id
    );
};
```

Note: `add_ior_component_to_profile()` is currently unimplemented.

The sample application's IOR interceptor implements `establish_components()` to perform the following tasks on an object reference's profile:

- Get its password policy.
- Set a `TAG_REQUIRES_PASSWORD` component accordingly.

Example 84: *Implementing establish_components()*

```
package demos.portable_interceptor.access_control.acl_service;

import org.omg.CORBA.*;
import org.omg.PortableInterceptor.*;
import org.omg.IOP.*;
import org.omg.IOP.CodecPackage.InvalidTypeForEncoding;

import demos.portable_interceptor.access_control.acl_service.*

1 class ACLIORInterceptorImpl
  extends LocalObject
  implements IORInterceptor
{
  ACLIORInterceptorImpl(Codec codec)
  {
    m_codec = codec;
  }

  public String name()
  {
    return NAME;
  }
```

Example 84: *Implementing establish_components()*

```
public void establish_components(IORInfo ior_info)
{
  AccessControl.PasswordPolicy pwd_policy = null;
  try {

2   Policy policy =
      ior_info.get_effective_policy(
        AccessControl.PASSWORD_POLICY_ID.value);
    pwd_policy =
      AccessControl.PasswordPolicyHelper.narrow(policy);
  }
  catch (INV_POLICY iv) {
    // PasswordPolicy wasn't set - return immediately
    return;
  }
  catch (BAD_PARAM ex) {
    ex.printStackTrace();
    System.exit(1);
  }

  Any cmpnt_data_any = ORB.init().create_any();

3

 cmpnt_data_any.insert_boolean(pwd_policy.requires_password())
 ;
  byte[] cmpnt_data = null;
  try {
4   cmpnt_data = m_codec.encode_value(cmpnt_data_any);
  }
  catch (InvalidTypeForEncoding ex) {
    ex.printStackTrace();
    System.exit(1);
  }

  // add TAG_REQUIRES_PASSWORD component to all profiles
5 TaggedComponent component = new TaggedComponent(
    AccessControlService.TAG_REQUIRES_PASSWORD.value,
    cmpnt_data);

6 ior_info.add_ior_component(component);
}
```

The sample application's implementation of `establish_components()` executes as follows:

1. Extends `org.omg.CORBA.LocalObject` because the IOR interceptor is a local object.
2. Gets the effective password policy object for the POA by calling `get_effective_policy()` on the `IORInfo`.
3. Gets the password policy value by calling `requires_password()` on the policy object.
4. Encodes the password policy value as an octet.
5. Instantiates a tagged component (`IOP::TaggedComponent`) and initializes it with the `TAG_REQUIRES_PASSWORD` tag and encoded password policy value.
6. Adds the tagged component to the object reference's profile by calling `add_ior_component()`.

Using RequestInfo Objects

Interception points for client and server interceptors receive `ClientRequestInfo` and `ServerRequestInfo` objects, respectively. These derive from `PortableInterceptor::RequestInfo`, which defines operations and attributes common to both.

Interface definition

The `RequestInfo` interface is defined as follows:

Example 85: *RequestInfo interface*

```
local interface RequestInfo {
    readonly attribute unsigned long request_id;
    readonly attribute string operation;
    readonly attribute Dynamic::ParameterList arguments;
    readonly attribute Dynamic::ExceptionList exceptions;
    readonly attribute Dynamic::ContextList contexts;
    readonly attribute Dynamic::RequestContext operation_context;
    readonly attribute any result;
    readonly attribute boolean response_expected;
    readonly attribute Messaging::SyncScope sync_scope;
    readonly attribute ReplyStatus reply_status;
    readonly attribute Object forward_reference;
    any get_slot (in SlotId id) raises (InvalidSlot);
    IOP::ServiceContext get_request_service_context (
        in IOP::ServiceId id);
    IOP::ServiceContext get_reply_service_context (
        in IOP::ServiceId id);
};
```

A `RequestInfo` object provides access to much of the information that an interceptor requires to evaluate a request and its service context data. For a full description of all attributes and operations, see the *CORBA Programmer's Reference*.

The validity of any given `RequestInfo` operation and attribute varies among client and server interception points. For example, the `result` attribute is valid only for interception points `receive_reply` on a client interceptor; and `send_reply` on a server interceptor. It is invalid for all other interception

points. Table 19 on page 543 and Table 20 on page 556 show which `RequestInfo` operations and attributes are valid for a given interception point.

Note: The Java implementation throws a `NO_RESOURCES` exception for the following attributes: `arguments`, `exceptions`, `contexts`, `operation_context`, and `result`.

Timeout attributes

A client might specify one or more timout policies on request or reply delivery. If portable interceptors are present in the bindings, these interceptors must be aware of the relevant timeouts so that they can bound any potentially blocking activities that they undertake.

The current OMG specification for portable interceptors does not account for timeout policy constraints; consequently, Orbix provides its own derivation of the `RequestInfo` interface, `IT_PortableInterceptor::RequestInfo`, which adds two attributes:

Example 86: *IT_PortableInterceptor::RequestInfo interface attributes*

```
module IT_PortableInterceptor
{
  local interface RequestInfo : PortableInterceptor::RequestInfo
  {
    readonly attribute TimeBase::UtcT request_end_time;
    readonly attribute TimeBase::UtcT reply_end_time;
  };
};
```

To access timeout constraints, interception point implementations can narrow their `ClientRequestInfo` or `ServerRequestInfo` objects to this interface. The two attributes apply to different interception points, as follows:

Table 18: *Portable Interceptor Timeout Attributes*

Timeout attribute	Relevant interception points

`request_end_time`	`send_request`
	`send_poll`
	`receive_request_service_contexts`
	`receive_request`

Table 18: *Portable Interceptor Timeout Attributes*

Timeout attribute	Relevant interception points

`reply_end_time`	`send_reply`
	`send_exception`
	`send_other`
	`receive_reply`
	`receive_exception`
	`receive_other`

Writing Client Interceptors

Interception point definitions

Client interceptors implement the `ClientRequestInterceptor` interface, which defines five interception points:

Example 87: *ClientRequestInterceptor interface*

```
local interface ClientRequestInterceptor : Interceptor {
    void send_request (in ClientRequestInfo ri)
        raises (ForwardRequest);
    void send_poll (in ClientRequestInfo ri);
    void receive_reply (in ClientRequestInfo ri);
    void receive_exception (in ClientRequestInfo ri)
        raises (ForwardRequest);
    void receive_other (in ClientRequestInfo ri)
        raises (ForwardRequest);
};
```

A client interceptor implements one or more of these operations.

In the password service example, the client interceptor provides an implementation for `send_request`, which encodes the required password in a service context and adds the service context to the object reference. For implementation details, see "Client Interceptor Tasks" on page 545.

Client interceptor constructor

As noted earlier, the ORB initializer instantiates and registers the client interceptor. This interceptor's constructor is implemented as follows:

Example 88: *Client interceptor constructor*

```
class ACLClientRequestInterceptorImpl
  extends LocalObject
  implements ClientRequestInterceptor
{
  ACLClientRequestInterceptorImpl(int password_slot, Codec
   codec)
  {
    m_password_slot = password_slot;
    m_codec = codec;
  }
  // ...
```

Client interceptor arguments

The client interceptor takes two arguments:

- The PICurrent slot allocated by the ORB initializer to store password data.
- An `IOP::Codec`, which is used to encode password data for service context data.

Interception Points

A client interceptor implements one or more interception points. During a successful request-reply sequence, each client-side interceptor executes one starting interception point and one ending interception point.

Starting interception points

Depending on the nature of the request, the ORB calls one of the following starting interception points:

send_request lets an interceptor query a synchronously invoked request, and modify its service context data before the request is sent to the server.

send_poll lets an interceptor query an asynchronously invoked request, where the client polls for a reply. This interception point currently applies only to deferred synchronous operation calls (see "Invoking Deferred Synchronous Requests" on page 382)

Ending interception points

Before the client receives a reply to a given request, the ORB executes one of the following ending interception points on that reply:

receive_reply lets an interceptor query information on a reply after it is returned from the server and before control returns to the client.

receive_exception is called when an exception occurs. It lets an interceptor query exception data before it is thrown to the client.

receive_other lets an interceptor query information that is available when a request results in something other than a normal reply or an exception. For example: a request can result in a retry, as when a GIOP reply with a `LOCATION_FORWARD` status is received; `receive_other` is also called on asynchronous calls, where the client resumes control before it receives a reply on a given request and an ending interception point is called.

Interception Point Flow

For each request-reply sequence, only one starting interception point and one ending point is called on a client interceptor. Each completed starting point is paired to an ending point. For example, if `send_request` executes to completion without throwing an exception, the ORB calls one of its ending interception points—`receive_reply`, `receive_exception`, or `receive_other`.

If multiple interceptors are registered on a client, the interceptors are traversed in order for outgoing requests, and in reverse order for incoming replies.

Scenario 1: Request-reply sequence is successful

Interception points A and B are registered with the server ORB. The interception point flow shown in Figure 39 depicts a successful reply-request sequence, where the server returns a normal reply:

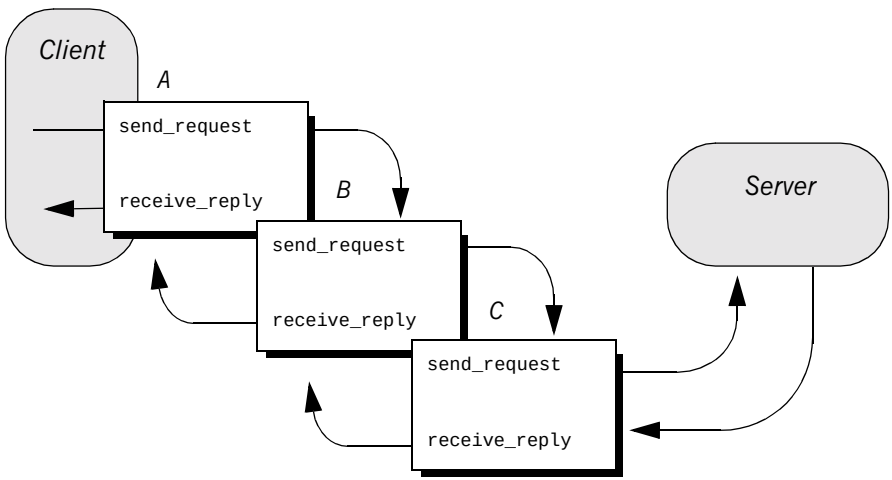


Figure 39: *Client interceptors process a normal reply.*

Scenario 2: Client receives LOCATION_FORWARD

If the server throws an exception or returns some other reply, such as LOCATION_FORWARD, the ORB directs the reply flow to the appropriate interception points, as shown in Figure 40:

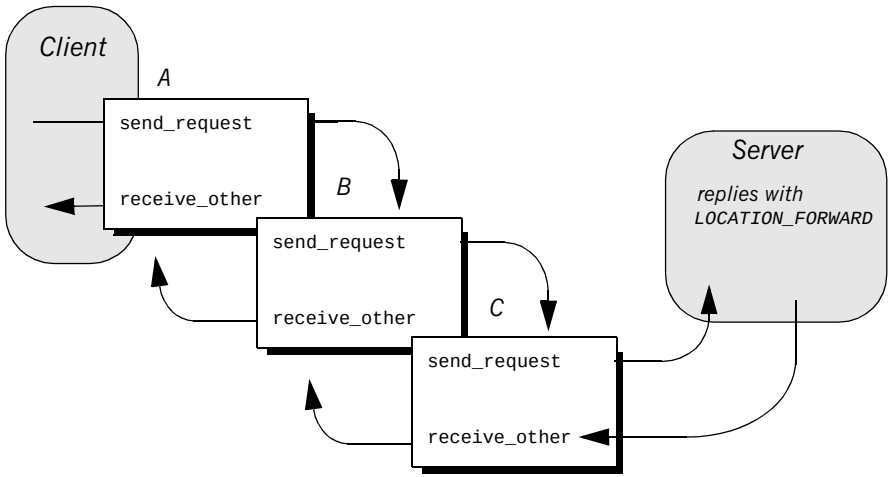


Figure 40: *Client interceptors process a LOCATION_FORWARD reply.*

Scenario 3: Exception aborts interception flow

Any number of events can abort or shorten the interception flow. Figure 41 shows the following interception flow:

1. Interceptor B's `send_request` throws an exception.
2. Because interceptor B's start point does not complete, no end point is called on it, and interceptor C is never called. Instead, the request flow returns to interceptor A's `receive_exception` end point.

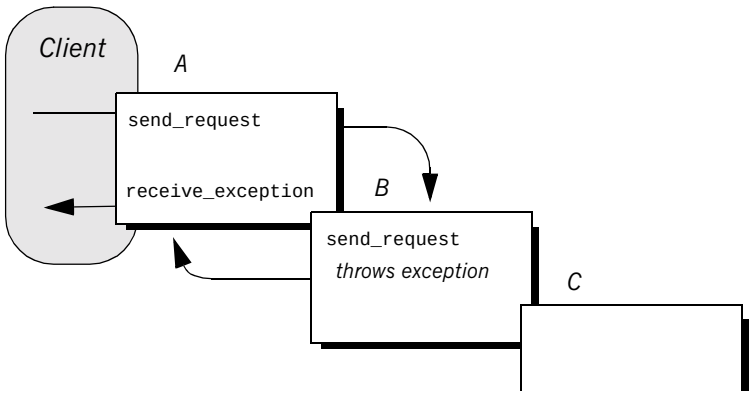


Figure 41: *send_request throws an exception in a client-side interceptor*

Scenario 4: Interceptor changes reply

An interceptor can change a normal reply to a system exception; it can also change the exception it receives, whether user or system exception to a different system exception. Figure 42 shows the following interception flow:

1. The server returns a normal reply.
2. The ORB calls `receive_reply` on interceptor C.
3. Interceptor C's `receive_reply` raises exception `foo_x`, which the ORB delivers to interceptor B's `receive_exception`.
4. Interceptor B's `receive_exception` changes exception `foo_x` to exception `foo_y`.
5. Interceptor A's `receive_exception` receives exception `foo_y` and returns it to the client.

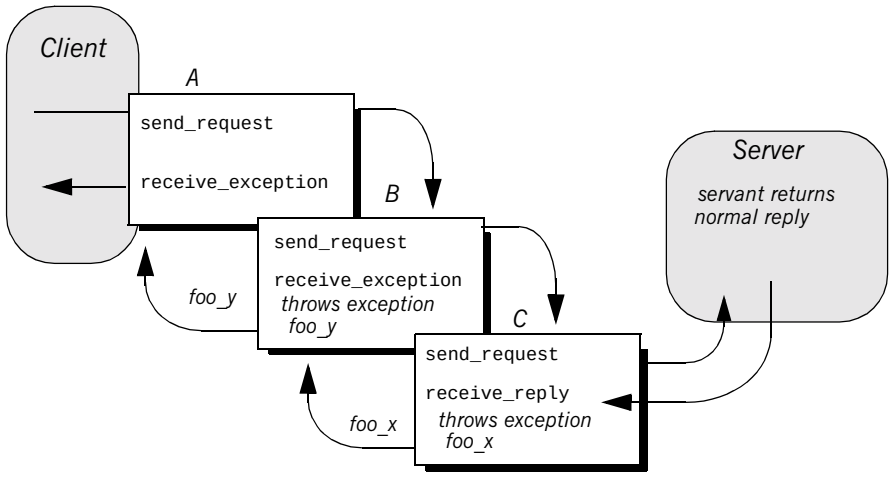


Figure 42: *Client interceptors can change the nature of the reply.*

Note: Interceptors must never change the CompletionStatus of the received exception.

ClientRequestInfo

Each client interception point gets a single `ClientRequestInfo` argument, which provides the necessary hooks to access and modify client request data:

Example 89: *ClientRequestInfo interface*

```
local interface ClientRequestInfo : RequestInfo {
    readonly attribute Object              target;
    readonly attribute Object              effective_target;
    readonly attribute IOP::TaggedProfile  effective_profile;
    readonly attribute any                 received_exception;
   readonly attribute CORBA::RepositoryId received_exception_id;

    IOP::TaggedComponent
    get_effective_component(in IOP::ComponentId id);

    IOP::TaggedComponentSeq
    get_effective_components(in IOP::ComponentId id);

    CORBA::Policy
    get_request_policy(in CORBA::PolicyType type);

    void
    add_request_service_context(
        in IOP::ServiceContext service_context,
        in boolean             replace
    );
};
```

Table 19 shows which `ClientRequestInfo` operations and attributes are accessible to each client interception point. In general, attempts to access an attribute or operation that is invalid for a given interception point throw an exception of `BAD_INV_ORDER` with a standard minor code of 10.

Table 19: *Client Interception Point Access to ClientRequestInfo*

ClientRequestInfo:	s_req	s_poll	r_reply	r_exep	r_other
request_id	y	y	y	y	y
operation	y	y	y	y	y
arguments	y[a]		y		
exceptions	y		y	y	y
contexts	y		y	y	y
operation_context	y		y	y	y
result			y		
response_expected	y	y	y	y	y
sync_scope	y		y	y	y
reply_status			y	y	y
forward_reference					y[b]
get_slot	y	y	y	y	y
get_request_service_context	y		y	y	y
get_reply_service_context			y	y	y
target	y	y	y	y	y
effective_target	y	y	y	y	y
effective_profile	y	y	y	y	y
received_exception				y	
received_exception_id				y	
get_effective_component	y		y	y	y

Table 19: *Client Interception Point Access to ClientRequestInfo*

ClientRequestInfo:	s_req	s_poll	r_reply	r_exep	r_other
get_effective_components	y		y	y	y
get_request_policy	y		y	y	y
add_request_service_context	y				

a. When `ClientRequestInfo` is passed to `send_request`, the arguments list contains an entry for all arguments, but only in and inout arguments are available.

b. Access to `forward_reference` is valid only if `reply_status` is set to `LOCATION_FORWARD` or `LOCATION_FORWARD_PERMANENT`.

Client Interceptor Tasks

A client interceptor typically uses a `ClientRequestInfo` to perform the following tasks:

- Evaluate an object reference's tagged components to determine an outgoing request's service requirements.
- Obtain service data from PICurrent.
- Encode service data as a service context.
- Add service contexts to a request.

These tasks are usually implemented in `send_request`. Interceptors have a much wider range of potential actions available to them—for example, client interceptors can call `get_request_service_context()`, to evaluate the service contexts that preceding interceptors added to a request. Other operations are specific to reply data or exceptions, and therefore can be invoked only by the appropriate `receive_` interception points.

This discussion confines itself to `send_request` and the tasks that it typically performs. For a full description of other `ClientRequestInfo` operations and attributes, see the *CORBA Programmer's Reference*.

In the sample application, the client interceptor provides an implementation for `send_request`, which performs these tasks:

- Evaluates each outgoing request for this tagged component to determine whether the request requires a password.
- Obtains service data from PICurrent
- Encodes the required password in a service context
- Adds the service context to the object reference:

Evaluating tagged components

The sample application's implementation of `send_request` checks each outgoing request for tagged component `TAG_REQUIRES_PASSWORD` by calling `get_effective_component()` on the interceptor's `ClientRequestInfo`:

Example 90: *Using get_effective_component()*

```
public void send_request(ClientRequestInfo request_info)
{
```

Example 90: *Using get_effective_component()*

```
1   if (requires_password(request_info))
    { // ...
    }

// ...

private boolean requires_password(ClientRequestInfo
   request_info)
{
  // check if a TAG_REQUIRES_PASSWORD component is present in the
  // effective profile
  //
  TaggedComponent password_component = null;
  try {
2   password_component = request_info.get_effective_component(
      TAG_REQUIRES_PASSWORD.value);
  }
  catch (BAD_PARAM bp) {
    // TAG_REQUIRES_PASSWORD component not present; treat as not
    // requiring a password
    return false;
  }

  // decode component data
  Any password_required_any = null;
  try {
3   password_required_any = m_codec.decode_value(
      password_component.component_data,
      ORB.init().get_primitive_tc(TCKind.tk_boolean));
  }
  catch (FormatMismatch ex) {
    ex.printStackTrace();
    System.exit(1);
  }
  catch (TypeMismatch ex) {
    ex.printStackTrace();
    System.exit(1);
  }

4 return password_required_any.extract_boolean();
}
```

The interception point executes as follows:

1. Calls the private method `require_password()` to determine whether a password is required.
2. `get_effective_component()` returns tagged component `TAG_REQUIRES_PASSWORD` from the request's object reference.
3. `decode_value()` is called on the interceptor's `Codec` to decode the octet sequence into a `CORBA::Any`. The call extracts the Boolean data that is embedded in the octet sequence.
4. The `Any`'s Boolean value is extracted and returned to `send_request()`.

Obtaining service data

After the client interceptor verifies that the request requires a password, it calls `RequestInfo::get_slot()` to obtain the client password from the appropriate slot:

Example 91: *Calling RequestInfo::get_slot()*

```
org.omg.CORBA.Any password_any = null;
try {
  password_any = request_info.get_slot(m_password_slot);
}
catch (InvalidSlot ex) {
  ex.printStackTrace();
  System.exit(1);
}
```

Encoding service context data

After the client interceptor gets the password string, it must convert the string and related data into a CDR encapsulation, so it can be embedded in a service context that is added to the request. To perform the data conversion, it calls `encode_value` on an `IOP::Codec`:

Example 92: *Calling IOP::Codec::encode_value()*

```
byte[] password_context_data = null;
try {
  password_context_data = m_codec.encode_value(password_any);
}
catch (InvalidTypeForEncoding ex) {
  ex.printStackTrace();
  System.exit(1);
}
```

Adding service contexts to a request

After initializing the service context, the client interceptor adds it to the outgoing request by calling `add_request_service_context()`:

Example 93: *Calling add_request_service_context()*

```
ServiceContext password_service_context = new ServiceContext(
  PASSWORD_SERVICE_ID.value, password_context_data);

// add service context to the request
request_info.add_request_service_context(
  password_service_context, true);
```

Writing Server Interceptors

Server interceptors implement the `ServerRequestInterceptor` interface:

Example 94: *ServerRequestInterceptor interface*

```
local interface ServerRequestInterceptor : Interceptor {
    void
    receive_request_service_contexts(in ServerRequestInfo ri
    ) raises (ForwardRequest);

    void
    receive_request(in ServerRequestInfo ri
    ) raises (ForwardRequest);

    void
    send_reply(in ServerRequestInfo ri);

    void
    send_exception(in ServerRequestInfo ri
    ) raises (ForwardRequest);

    void
    send_other(in ServerRequestInfo ri
    ) raises (ForwardRequest);
};
```

Interception Points

During a successful request-reply sequence, each server interceptor executes one starting interception point and one intermediate interception point for incoming requests. For outgoing replies, a server interceptor executes an ending interception point.

Starting interception point

A server interceptor has a single starting interception point:

receive_request_service_contexts lets interceptors get service context information from an incoming request and transfer it to PICurrent slots. This interception point is called before the servant manager is called. Operation parameters are not yet available at this point.

Intermediate interception point

A server interceptor has a single intermediate interception point:

receive_request lets an interceptor query request information after all information, including operation parameters, is available.

Ending interception points

An ending interception point is called after the target operation is invoked, and before the reply returns to the client. The ORB executes one of the following ending interception points, depending on the nature of the reply:

send_reply lets an interceptor query reply information and modify the reply service context after the target operation is invoked and before the reply returns to the client.

send_exception is called when an exception occurs. An interceptor can query exception information and modify the reply service context before the exception is thrown to the client.

send_other lets an interceptor query the information available when a request results in something other than a normal reply or an exception. For example, a request can result in a retry, as when a GIOP reply with a `LOCATION_FORWARD` status is received.

Interception Point Flow

For a given server interceptor, the flow of execution follows one of these paths:

- `receive_request_service_contexts` completes execution without throwing an exception. The ORB calls that interceptor's intermediate and ending interception points. If the intermediate point throws an exception, the ending point for that interceptor is called with the exception.
- `receive_request_service_contexts` throws an exception. The interceptor's intermediate and ending points are not called.

If multiple interceptors are registered on a server, the interceptors are traversed in order for incoming requests, and in reverse order for outgoing replies. If one interceptor in the chain throws an exception in either its starting or intermediate points, no other interceptors in the chain are called; and the appropriate ending points for that interceptor and all preceding interceptors are called.

Scenario 1: Target object throws exception

Interceptors A and B are registered with the server ORB. Figure 43 shows the following interception flow:

1. The interception point `receive_request_server_contexts` processes an incoming request on interceptor A, then B. Neither interception point throws an exception.
2. Intermediate interception point `receive_request` processes the request first on interceptor A, then B. Neither interception point throws an exception.
3. The ORB delivers the request to the target object. The object throws an exception.
4. The ORB calls interception point `send_exception`, first on interceptor B., then A, to handle the exception.

5. The ORB returns the exception to the client.

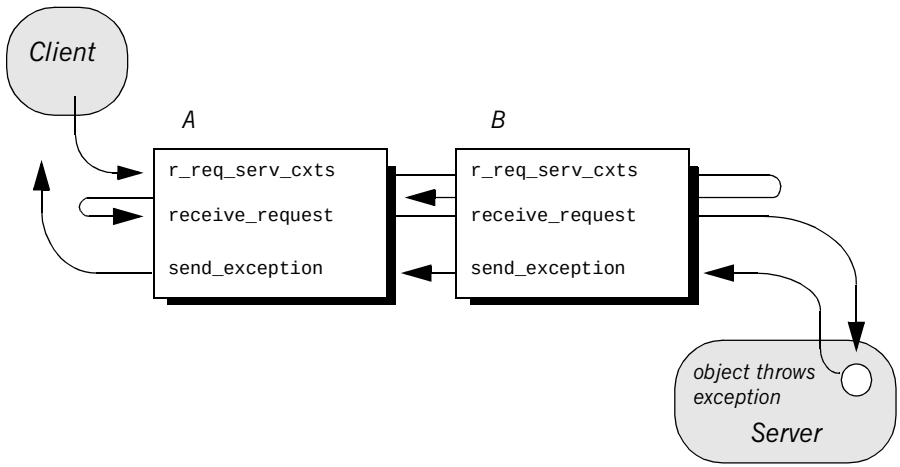


Figure 43: *Server interceptors receive request and send exception thrown by target object.*

Scenario 2: Exception aborts interception flow

Any number of events can abort interception flow. Figure 44 shows the following interception flow.

1. A request starts server-side interceptor processing, starting with interceptor A's `receive_request_service_contexts`. The request is passed on to interceptor B.
2. Interceptor B's `receive_request_service_contexts` throws an exception. The ORB aborts interceptor flow and returns the exception to interceptor A's end interception point `send_exception`.
3. The exception is returned to the client.

Because interceptor B's start point does not complete execution, its intermediate and end points are not called. Interceptor A's intermediate point `receive_request` also is not called.

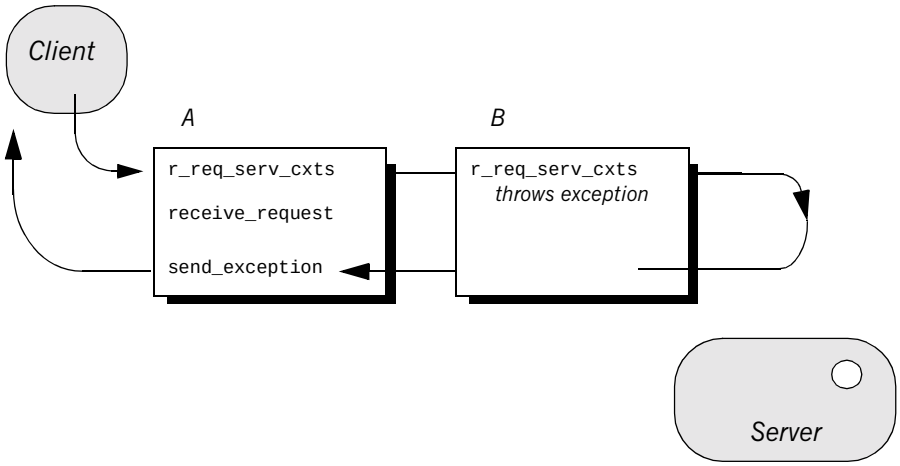


Figure 44: *receive_request_service_contexts throws an exception and interception flow is aborted.*

Scenario 3: Interceptors change reply type

An interceptor can change a normal reply to a system exception; it can also change the exception it receives, whether user or system exception to a different system exception. Figure 45 shows the following interception flow:

1. The target object returns a normal reply.
2. The ORB calls `send_reply` on server interceptor C.
3. Interceptor C's `send_reply` interception point throws exception `foo_x`, which the ORB delivers to interceptor B's `send_exception`.
4. Interceptor B's `send_exception` changes exception `foo_x` to exception `foo_y`, which the ORB delivers to interceptor A's `send_exception`.
5. Interceptor A's `send_exception` returns exception `foo_y` to the client.

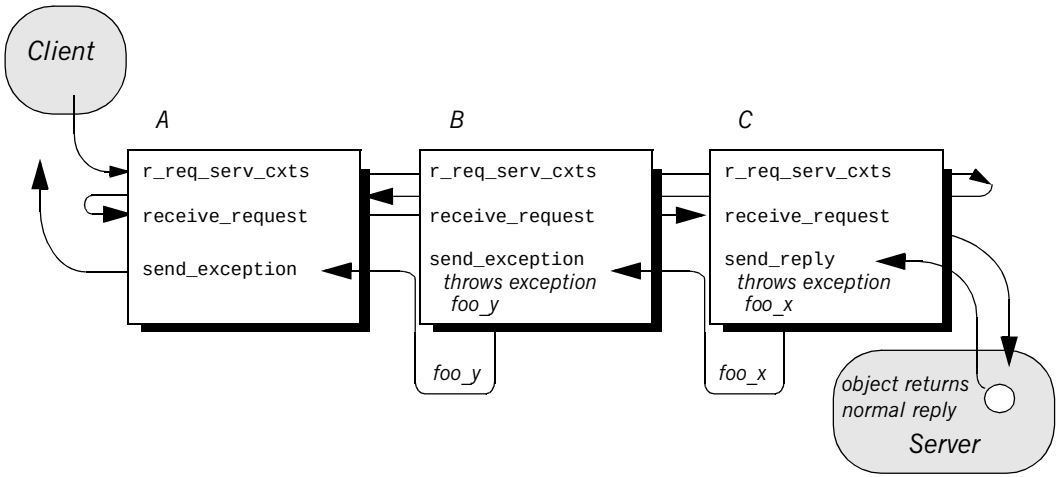


Figure 45: *Server interceptors can change the reply type.*

Note: Interceptors must never change the CompletionStatus of the received exception.

ServerRequestInfo

Each server interception point gets a single `ServerRequestInfo` argument, which provides the necessary hooks to access and modify server request data:

Example 95: *ServerRequestInfo interface*

```
local interface ServerRequestInfo : RequestInfo {
    readonly attribute any sending_exception;
    readonly attribute CORBA::OctetSeq object_id;
    readonly attribute CORBA::OctetSeq adapter_id;
    readonly attribute CORBA::RepositoryId
        target_most_derived_interface;


    CORBA::Policy
    get_server_policy(in CORBA::PolicyType type);

    void
    set_slot(
        in SlotId id,
        in any    data
    ) raises (InvalidSlot);

    boolean
    target_is_a(in CORBA::RepositoryId id);

    void
    add_reply_service_context(
        in IOP::ServiceContext service_context,
        in boolean             replease
    );
};
```

Table 20 shows which `ServerRequestInfo` operations and attributes are accessible to server interception points. In general, attempts to access an attribute or operation that is invalid for a given interception point raise an exception of `BAD_INV_ORDER` with a standard minor code of 10.

Table 20: *Server Interception Point Access to ServerRequestInfo*

ServerRequestInfo:	r_req_ serv_cxts	r_req	s_reply	s_excep	s_other
request_id	y	y	y	y	y
operation	y	y	y	y	y
arguments[a]	y	y	y		
exceptions		y	y	y	y
contexts		y	y	y	y
operation_context		y	y		
result			y		
response_expected	y	y	y	y	y
sync_scope	y	y	y	y	y
reply_status			y	y	y
forward_reference					y
get_slot	y	y	y	y	y
get_request_service_context	y	y	y	y	y
get_reply_service_context			y	y	y
sending_exception				y	
object_id		y			
adapter_id		y			
target_most_derived_interface		y			
get_server_policy	y	y	y	y	y

Table 20: *Server Interception Point Access to ServerRequestInfo*

ServerRequestInfo:	r_req_ serv_cxts	r_req	s_reply	s_excep	s_other
set_slot	y	y	y	y	y
target_is_a		y			
add_reply_service_context	y	y	y	y	y

a. When a ServerRequestInfo is passed to receive_request(), the arguments list contains an entry for all arguments, but only in and inout arguments are available.

Server Interceptor Tasks

A server interceptor typically uses a `ServerRequestInfo` to perform the following tasks:

- Get server policies.
- Get service contexts from an incoming request and extract their data.

The sample application implements `receive_request_server_contexts` only. The requisite service context data is available at this interception point, so it is capable of executing authorizing or disqualifying incoming requests. Also, unnecessary overhead is avoided for unauthorized requests: by throwing an exception in `receive_request_server_contexts`, the starting interception point fails to complete and all other server interception points are bypassed.

This discussion confines itself to `receive_request_server_contexts` and the tasks that it typically performs. For a description of other `ServerRequestInfo` operations and attributes, see the *CORBA Programmer's Reference*.

Get server policies

The sample application's `receive_request_server_contexts` implementation obtains the server's password policy in order to compare it to the password that accompanies each request. In order to do so, it calls `get_server_policy()` on the interception point's `ServerRequestInfo`:

Example 96: *Calling get_server_policy()*

```
// ...
import demos.portable_interceptor.access_control.acl_service.*;

public void receive_request_service_contexts(
  ServerRequestInfo request_info)
{
  // determine whether password protection is required by
  // the effective policies
  AccessControl.PasswordPolicy password_policy = null;
  try {
    password_policy = PasswordPolicyHelper.narrow(
      request_info.get_server_policy(
        AccessControl.PASSWORD_POLICY_ID.value));
  }
```

Example 96: *Calling get_server_policy()*

```
  catch (INV_POLICY ex) {
    // password policy not set
    return;
  }
  catch (BAD_PARAM ex) {
    ex.printStackTrace();
    System.exit(1);
  }
// ...
```

Get service contexts

After `receive_request_server_contexts` gets the server's password policy, it needs to compare it to the client password that accompanies the request. The password is encoded as a service context, which is accessed through its identifier `PASSWORD_SERVICE_ID`:

Example 97:

```
  // ...
  if (password_policy != null
      && password_policy.requires_password())
  {
  // check that the correct password was sent with request
    if (!check_password(
         request_info, password_policy.password()))
    {
      throw new NO_PERMISSION(
        0xDEADBEEF, CompletionStatus.COMPLETED_NO);
    }
  }
  // ...

private boolean check_password(ServerRequestInfo request_info,
                               String expected_password)
{
  org.omg.IOP.ServiceContext password_service_context = null;
  try {
    // get the password service context ...
```

Example 97:

```
1     password_service_context =
        request_info.get_request_service_context(
          AccessControlService.PASSWORD_SERVICE_ID.value);
    }
    catch (BAD_PARAM bp) {
      // PASSWORD_SERVICE_ID service context not present in request
      return false;
    }

    // decode context data
    Any password_any = null;
    try {
2     password_any = m_codec.decode_value(
        password_service_context.context_data,
        ORB.init().get_primitive_tc(TCKind.tk_string));
    }
    catch (FormatMismatch ex) {
      ex.printStackTrace();
      System.exit(1);
    }
    catch (TypeMismatch ex) {
      ex.printStackTrace();
      System.exit(1);
      }

    // compare the passwords
3   String received_password = password_any.extract_string();
    return expected_password.equals(received_password);
}
```

The interception point executes as follows:

1. Calls `get_request_service_context()` with an argument of `AccessControlService::PASSWORD_SERVICE_ID`. If successful, the call returns with a service context that contains the client password.
2. Calls `decode_value()` on the interceptor's `Codec` to decode the service context data into a `CORBA::Any`. The call specifies to extract the string data that is embedded in the octet sequence.

3. Extracts the Any's string value and compares it to the server password. If the two strings match, the request passes authorization and is allowed to proceed; otherwise, an exception is thrown back to the client.

Registering Portable Interceptors

Portable interceptors and their components are instantiated and registered during ORB initialization, through an ORB initializer. An ORB initializer implements its `pre_init()` or `post_init()` operation, or both. The client and server applications must register the ORB initializer before calling `ORB_init()`.

Implementing an ORB Initializer

The sample application's ORB initializer implements `pre_init()` to perform these tasks:

- Obtain PICurrent and allocate a slot for password data.
- Encapsulate PICurrent and the password slot identifier in an `AccessControl::Current` object, and register this object with the ORB as an initial reference.
- Register a password policy factory.
- Create Codec objects for the application's interceptors, so they can encode and decode service context data and tagged components.
- Register interceptors with the ORB.

Obtain PICurrent

In the sample application, the client application and client interceptor use PICurrent to exchange password data:

- The client thread places the password in the specified PICurrent slot.
- The client interceptor accesses the slot to obtain the client password and add it to outgoing requests.

In the sample application, `pre_init()` calls the following operations on `ORBInitInfo`:

1. `allocate_slot_id()` allocates a slot and returns the slot's identifer.
2. `resolve_initial_references("PICurrent")` returns PICurrent.

Example 98: *Obtaining PICurrent*

```
public void pre_init(ORBInitInfo init_info)
{

  // reserve a slot for AccessControl::Current
1 int password_slot = init_info.allocate_slot_id();

  // get a reference to PICurrent
  org.omg.PortableInterceptor.Current pi_current = null;
  try {
```

Example 98: *Obtaining PICurrent*

```
2     org.omg.CORBA.Object obj
        init_info.resolve_initial_references("PICurrent");
      pi_current =
        org.omg.PortableInterceptor.CurrentHelper.narrow(obj);
    }
    catch (InvalidName ex) {
      ex.printStackTrace();
      System.exit(1);
    }
    catch (BAD_PARAM ex) {
      ex.printStackTrace();
      System.exit(1);
    }
    // ...
```

Register an initial reference

After the ORB initializer obtains PICurrent and a password slot, it must make this information available to the client thread. To do so, it instantiates an `AccessControl::Current` object. This object encapsulates:

- PICurrent and its password slot
- Operations that access slot data

The `AccessControl::Current` object has the following IDL definition:

Example 99: *AccessControl::Current interface*

```
module AccessControl {
    // ...
    local interface Current : CORBA::Current {
        attribute string password;
    };
};
```

The application implements `AccessControl::Current` as follows:

Example 100: *Implementing an AccessControl::Current object*

```
class ACLCurrentImpl
  extends LocalObject
  implements
   demos.portable_interceptor.access_control.acl_service.
   AccessControl.Current
```

Example 100:*Implementing an AccessControl::Current object*

```
{
  ACLCurrentImpl(org.omg.PortableInterceptor.Current pi_current,
                 int password_slot)
  {
    m_pi_current = pi_current;
    m_password_slot = password_slot;
  }

  public String password()
  {
    // get password from PICurrent slot
    Any password_any = null;
    try {
      password_any = m_pi_current.get_slot(m_password_slot);
    }
    catch (InvalidSlot ex) {
      ex.printStackTrace();
      System.exit(1);
    }

    return password_any.extract_string();
  }

  public void password(String password)
  {
    // set password in PICurrent slot
    try {
      System.out.println("setting password from PICurrrent
   slot");
      Any password_any = ORB.init().create_any();
      password_any.insert_string(password);
      m_pi_current.set_slot(m_password_slot, password_any);
    }
    catch (InvalidSlot ex) {
      ex.printStackTrace();
      System.exit(1);
    }
  }
  // ...
}
```

With `AccessControl::Current` thus defined, the ORB initializer performs these tasks:

1. Instantiates the `AccessControl::Current` object.
2. Registers it as an initial reference.

Example 101: *Registering AccessControl::Current as an intial reference*

```
try {
  demos.portable_interceptor.access_control.acl_service.
  AccessControl.Current acl_current =
    new ACLCurrentImpl(pi_current, password_slot);
  init_info.register_initial_reference(
    "AccessControlCurrent", acl_current);
}
catch (InvalidName ex) {
  ex.printStackTrace();
  System.exit(1);
}
```

Create and register policy factories

The sample application's IDL defines the following password policy to provide password protection for the server's POAs.

Example 102: *Defining a password policy*

```
module AccessControl {
    const CORBA::PolicyType PASSWORD_POLICY_ID = 0xBEEF;

    struct PasswordPolicyValue {
        boolean requires_password;
        string  password;
    };

    local interface PasswordPolicy : CORBA::Policy {
        readonly attribute boolean requires_password;
        readonly attribute string  password;
    };

    local interface Current : CORBA::Current {
        attribute string password;
    };
};
```

During ORB initialization, the ORB initializer instantiates and registers a factory for password policy creation:

```
PolicyFactory password_policy_factory =
  new PasswordPolicyFactoryImpl();
  init_info.register_policy_factory(
    AccessControl.PASSWORD_POLICY_ID.value,
    password_policy_factory);
```

For example, a server-side ORB initializer can register a factory to create a password policy, to provide password protection for the server's POAs.

Create Codec objects

Each portable interceptor in the sample application requires a `PortableInterceptor::Codec` in order to encode and decode octet data for service contexts or tagged components. The ORB initializer obtains a `Codec` factory by calling `ORBInitInfo::codec_factory`, then creates a `Codec`:

Example 103: *Creating a Codec object*

```
org.omg.IOP.Codec cdr_codec = null;
try {
  Encoding cdr_encoding = new Encoding(
    org.omg.IOP.ENCODING_CDR_ENCAPS.value, (byte)1, (byte)2);
  cdr_codec =
    init_info.codec_factory().create_codec(cdr_encoding);
}
catch (UnknownEncoding ex) {
  ex.printStackTrace();
  System.exit(1);
}
```

When the ORB initializer instantiates portable interceptors, it supplies this `Codec` to the interceptor constructors.

Register interceptors

The sample application relies on three interceptors:

- An IOR interceptor that adds a `TAG_PASSWORD_REQUIRED` component to IOR's that are generated by the server application.
- A client interceptor that attaches a password as a service context to outgoing requests.

- A server interceptor that checks a request's password before allowing it to continue.

Note: The order in which the ORB initializer registers interceptors has no effect on their runtime ordering. The order in which portable initializers are called is determined by their order in the client and server binding lists (see "Setting Up Orbix to Use Portable Interceptors" on page 570)

The ORB initializer instantiates and registers these interceptors as follows:

Example 104: *Registering interceptors*

```
// Register IOR interceptor
try {
  IORInterceptor ior_interceptor =
    new ACLIORInterceptorImpl(cdr_codec);
  init_info.add_ior_interceptor(ior_interceptor);
}
  catch (DuplicateName ex) {
      ex.printStackTrace();
      System.exit(1);
}

// Register client interceptor
try {
  ClientRequestInterceptor client_interceptor =
    new ACLClientRequestInterceptorImpl(password_slot,
   cdr_codec);
  init_info.add_client_request_interceptor(client_interceptor);
}
catch (DuplicateName ex) {
  ex.printStackTrace();
  System.exit(1);
}

// Register server interceptor
try {
  ServerRequestInterceptor server_interceptor =
    new ACLServerRequestInterceptorImpl(cdr_codec);

   init_info.add_server_request_interceptor(server_interceptor);
}
catch (DuplicateName ex) {
  ex.printStackTrace();
  System.exit(1);
}
```

Registering an ORBInitializer

An application registers an ORB initializer via JAVA ORB properties as follows:

`org.omg.PortableInterceptor.ORBInitializerClass.`*`Service`*

`Service` is the string name of a class that implements `org.omg.PortableInterceptor.ORBInitializer`. During initialization of a new ORB (an ORB with a unique identifier), ORB initializers are registered in the following steps:

1. All `org.omg.PortableInterceptor.ORBInitializerClass` ORB properties are collected and the *`Service`* string is extracted.
2. An object is instantiated with *`Service`* as its class name.
3. The ORB intializer's `pre_init` and `post_init` methods are called.

Setting Up Orbix to Use Portable Interceptors

The following setup requirements apply to registering portable interceptors with the Orbix configuration. At the appropriate scope, add:

- `portable_interceptor` plugin to `orb_plugins`.
- Client interceptor names to `client_binding_list`.
- Server interceptor names to `server_binding_list`.

You can only register portable interceptors for ORBs created in programs that are linked with the shared library `it_portable_interceptor`. If an application has unnamed (anonymous) portable interceptors, add `AnonymousPortableInterceptor` to the client and server binding lists. All unnamed portable interceptors insert themselves at that location in the list.

Note: The binding lists determine the order in which interceptors are called during request processing.

For more information about Orbix configuration, see the *Application Server Platform Administrator's Guide*.

CHAPTER 20

Bidirectional GIOP

The usual GIOP connection semantics allow request messages to be sent in only one direction over a connection-oriented transport protocol. Recent changes to the GIOP standard allow this restriction to be relaxed in certain circumstances, making it possible to use connections in a bidirectional mode.

In this chapter

This chapter contains the following sections:

Introduction to Bidirectional GIOP

Overview

The original OMG General Inter-ORB Protocol (GIOP) standard specified that client/server connections are *unidirectional*, in the sense that GIOP request messages can be sent in one direction only (from client to server).

There are certain scenarios, however, where it is important to lift the unidirectional constraint on connections. For example, when a client connects to a server through a firewall, it is usually impossible for the server to open a new TCP/IP connection back to the client. In this scenario, the only feasible option is to re-use the existing incoming connection by making it *bidirectional*.

Bidirectional GIOP draft specification

At the time of writing, a draft specification for bidirectional GIOP is described in the OMG firewall submission:

`http://www.omg.org/docs/orbos/01-08-03.pdf`

Features

IONA's implementation of bidirectional GIOP has the following features:

1. Compliant with the modified bidirectional GIOP approach described in the firewall submission.
2. Compatible with GIOP 1.2 (that is, not dependent on GIOP 1.4 `NegotiateSession` messages).
3. Decoupled from IIOP, so that it can be used over arbitrary connection-oriented transports (for example, SHMIOP).
4. Supports weak `BiDirIds` initially.
5. Supports bidirectional invocations on legacy Orbix 3.x callback object references in order to facilitate phased migration to Orbix 6.1.

Configuration versus programming approach

There are essentially two alternative approaches you can take to enabling bidirectional GIOP in your Orbix applications, as follows:

- Configuration approach.
- Programming approach.

Configuration approach

The configuration approach to enabling bidirectional GIOP has the advantage of being relatively easy to do, because it does not require an application re-build.

On the other hand, this approach has the disadvantage that it is coarse grained: that is, the relevant bidirectional policies are applied to *all* of the CORBA objects, object references and POA instances.

For details of this approach, see the *Orbix Administrator's Guide*.

Programming approach

The programming approach to enabling bidirectional GIOP has the advantage that you can apply it at any level of granularity: ORB, POA, thread or object. In general, it is better to apply a fine-grained approach—that is, enabling bidirectional GIOP only for those objects that really need it.

Bidirectional GIOP incurs a small performance penalty, due to the following overheads: extra component added to IORs, extra service context added to request messages, checking for bidirectional policy compatibility. By enabling bidirectional GIOP only where it is needed, you can minimize this performance penalty.

Bidirectional GIOP Policies

Overview

Bidirectional GIOP is enabled and controlled by setting a variety of CORBA policies. The bidirectional policies are defined by two different IDL modules, as follows:

- IDL for standard policies—defined by the OMG.
- IDL for proprietary policies—defined by IONA.

IDL for standard policies

The OMG draft specification for bidirectional GIOP defines three bidirectional policies. These policies are defined in the `BiDirPolicy` IDL module as shown in Example 105.

Example 105: *The BiDirPolicy Module*

```
// IDL
module BiDirPolicy
{
    typedef unsigned short BidirectionalPolicyValue;

    const BidirectionalPolicyValue ALLOW = 0;
    const BidirectionalPolicyValue DENY  = 1;

    // to be assigned by OMG (using temporary IDs
    // allocated from IONA namespace)
    //
    const CORBA::PolicyType BI_DIR_EXPORT_POLICY_TYPE =
   0x49545F7C;
    const CORBA::PolicyType BI_DIR_OFFER_POLICY_TYPE =
   0x49545F7D;
    const CORBA::PolicyType BI_DIR_ACCEPT_POLICY_TYPE =
   0x49545F7E;

    local interface BidirectionalExportPolicy : CORBA::Policy
    {
        readonly attribute BidirectionalPolicyValue value;
    };

    local interface BidirectionalOfferPolicy : CORBA::Policy
    {
        readonly attribute BidirectionalPolicyValue value;
    };
```

Example 105: *The BiDirPolicy Module*

```
    local interface BidirectionalAcceptPolicy : CORBA::Policy
    {
        readonly attribute BidirectionalPolicyValue value;
    };
};
```

BidirectionalExportPolicy

The `BiDirPolicy::BidirectionalExportPolicy` is a policy that is applied to POA instances on the client side (in this context, the term *client* here designates the process that opens the bidirectional connection). There are two alternative values for this policy:

- `BiDirPolicy::ALLOW`—indicates that the CORBA objects activated by this POA are able to receive callbacks through a bidirectional GIOP connection.
- `BiDirPolicy::DENY` (the default)—the bidirectional export policy is disabled.

In practice, when the `BidirectionalExportPolicy` is enabled on a POA instance, an ID, `GIOP::BiDirId`, is generated for the POA. The `BiDirId` is used to identify the POA in the context of managing bidirectional connections. In particular, the `BiDirId` is embedded in IORs generated by this POA (encoded in a `TAG_BI_DIR_GIOP` IOR component).

BidirectionalOfferPolicy

The `BiDirPolicy::BidirectionalOfferPolicy` is a policy that can be applied to object references on the client side (that is, object references whose operations are invoked by the client, not callback object references created by the client). There are two alternative values for this policy:

- `BiDirPolicy::ALLOW`—indicates that the outgoing connection used by this object reference will be offered as a bidirectional GIOP connection.
- `BiDirPolicy::DENY` (the default)—the bidirectional offer policy is disabled.

The mechanism for making a bidirectional offer is based on sending a list of `BiDirId`'s in a `GIOP::BI_DIR_GIOP_OFFER` service context. Hence, the bidirectional offer is not made until you invoke an operation on the offer-enabled object reference.

BidirectionalAcceptPolicy

The `BiDirPolicy::BidirectionalAcceptPolicy` is a policy that can be applied to *callback object references* on the server side. Normally, the bidirectional accept policy should be overridden only on callback object references whose IOR could reasonably be expected to contain a `BiDirId` component—otherwise the bidirectional accept policy has no effect. There are two alternative values for this policy:

- `BiDirPolicy::ALLOW`—indicates that the callback object reference should attempt to re-use one of the incoming connections to send invocation requests back to the client.
- `BiDirPolicy::DENY` (the default)—the bidirectional accept policy is disabled.

When the server first invokes an operation on the callback object reference, Orbix extracts the `BiDirId` from the associated IOR and attempts to match this `BiDirId` with one of the offered incoming connections. Successful re-use of an incoming connection requires a `BiDirId` match and compatible policies.

IDL for proprietary policies

Orbix defines some proprietary bidirectional GIOP policies, in addition to the policies defined by the OMG draft specification. These policies are defined in the `IT_BiDirPolicy` IDL module as shown in Example 106.

Example 106: *The IT_BiDirPolicy Module*

```
// IDL
...
module IT_BiDirPolicy
{
    const CORBA::PolicyType BI_DIR_ID_GENERATION_POLICY_ID =
        IT_PolicyBase::IONA_POLICY_ID + 62;

    const CORBA::PolicyType BI_DIR_GEN3_ACCEPT_POLICY_ID =
        IT_PolicyBase::IONA_POLICY_ID + 65;

    typedef unsigned short BiDirIdGenerationPolicyValue;
    const BiDirIdGenerationPolicyValue RANDOM = 0;
    const BiDirIdGenerationPolicyValue REPEATABLE = 1;

    local interface BiDirIdGenerationPolicy : CORBA::Policy
    {
        readonly attribute BiDirIdGenerationPolicyValue value;
```

Example 106: *The IT_BiDirPolicy Module*

```
    };

    local interface BidirectionalGen3AcceptPolicy : CORBA::Policy
    {
        readonly attribute BiDirPolicy::BidirectionalPolicyValue
    value;
    };
};
```

BiDirIdGenerationPolicy

The `IT_BiDirPolicy::BiDirIdGenerationPolicy` is a proprietary policy that affects the way `GIOP::BiDirId`'s are generated. It is applied to POA instances on the client side and must be used in combination with the `BiDirPolicy::BidirectionalExportPolicy`. There are two alternative values for this policy:

- `IT_BiDirPolicy::RANDOM` (the default)—the `BidDirId` combines a 32-bit endpoint creation timestamp and 128 bit hash/digest of the endpoint ID. The use of the timestamp makes accidental clashes extremely unlikely.
- `IT_BiDirPolicy::REPEATABLE`—the `BiDirId` is composed entirely of a 160 bit hash/digest of the endpoint ID. Accidental clashes are possible if similar lengthy fully qualified POA names are extensively used in the same location domain, but the probability of a clash is still very low.

Note: If callback object references are intended to be persistent, the `REPEATABLE` policy value must be chosen to ensure that the same `BiDirId` is generated over subsequent re-activations of the client process. In the usual callback scenario, however, the callback object references are transient and the `RANDOM` policy value is applicable.

BidirectionalGen3AcceptPolicy

The `IT_BiDirPolicy::BidirectionalGen3AcceptPolicy` is a policy that can be applied to Orbix 3 callback object references on the server side. This policy is provided to facilitate interoperability between Orbix 6.x servers and Orbix 3 legacy clients. The effect of this policy is analogous to the `BidirectionalAcceptPolicy`, except that it applies to Orbix 3 callbacks.

There are two alternative values for this policy:

- `BiDirPolicy::ALLOW`—indicates that the Orbix 3 callback object reference should attempt to re-use one of the incoming connections to send invocation requests back to the Orbix 3 client.
- `BiDirPolicy::DENY` (the default)—the bidirectional Orbix 3 accept policy is disabled.

For more details on interoperability with Orbix 3, see "Interoperability with Orbix Generation 3" on page 595.

Policy granularity

As usual for CORBA policies, these bidirectional policies can be defined at different levels of granularity. The different levels of granularity for which you can define each policy are summarized in Table 21.

Table 21: *Levels of Granularity for Bidirectional Policies*

Bidirectional GIOP Policy	Levels of Granularity
`BiDirPolicy::BidirectionalExportPolicy`	ORB POA
`BiDirPolicy::BidirectionalOfferPolicy`	ORB Thread Object reference
`BiDirPolicy::BidirectionalAcceptPolicy`	ORB Thread Object reference
`IT_BiDirPolicy::BiDirIdGenerationPolicy`	ORB POA

Table 21: *Levels of Granularity for Bidirectional Policies*

Bidirectional GIOP Policy	Levels of Granularity
`IT_BiDirPolicy::BidirectionalGen3AcceptPolicy`	ORB Thread Object reference

Configuration Prerequisites

Overview

This subsection describes the basic configuration prerequisites for using bidirectional GIOP in an Orbix 6.x domain.

Note: You would normally not have to configure these configuration settings manually. In a generated configuration domain, by default, your client and server binding lists are set to include `BiDir_GIOP`.

Client configuration

On the client-side, the `plugins:giop:message_server_binding_list` should include an entry for `BiDir_GIOP`, for example:

```
plugins:giop:message_server_binding_list=
    ["BiDir_GIOP","GIOP" ];
```

This enables the existing outgoing message interceptor chain to be re-used for an incoming server binding.

Server configuration

On the server-side, the `binding:client_binding_list` should include an entry for `BiDir_GIOP`, for example:

```
binding:client_binding_list = [ "OTS+BiDir_GIOP", "BiDir_GIOP",
   "OTS+GIOP+IIOP", "GIOP+IIOP", ... ];
```

This enables the existing incoming message interceptor chain to be re-used, so that the outgoing client binding dispatches the callback invocation.

Note: If your server needs to interoperate with Orbix 3 legacy clients, the `binding:client_binding_list` should also include a `"BiDir_Gen3"` entry. See "Interoperability with Orbix Generation 3" on page 595.

Basic BiDir Scenario

Overview

This section describes the stock feed demonstration, which is a sample bidirectional GIOP scenario. Some code examples extracted from the stock feed demonstration show you how to set the bidirectional GIOP policies on the client side and on the server side.

In this section

This section contains the following subsections:

The Stock Feed Demonstration

Overview

This section describes the *stock feed* demonstration, a basic bidirectional GIOP scenario. The stock feed system consists of one central server, which gathers information about stock price changes, and many clients, which can register an interest in receiving stock data.

The central server stores a list of callback object references for all the clients that are registered with it. As soon as a stock price changes occurs, the server iterates over the list of callback object references, calling `NotifyPriceChange()` on each one. It is these callback invocations which can potentially be configured to use bidirectional GIOP.

Demonstration code

The stock feed demonstration code is located in the following directory:

OrbixInstallDir`/asp/`*Version*`/demos/corba/orb/bidir_giop`

IDL for stock feed scenario

Example 107 shows the IDL for the stock feed demonstration, which consists of two IDL interfaces: `StockInfoCB` and `RegStockInfo`. These IDL interfaces are identical to the ones used by the corresponding demonstration in the Orbix Generation 3 product.

Example 107: *IDL for the Stock Feed Demonstration*

```
// IDL
interface StockInfoCB
{
    oneway void NotifyPriceChange(
        in string stock_name,
        in float new_price
    );
};

interface RegStockInfo
{
    void Register(in StockInfoCB callback);
    void Deregister(in StockInfoCB callback);
    void Notify(in float new_price);
};
```

Stock feed scenario

Figure 46 gives you an overview of the stock feed demonstration, where a number of clients register their interest in receiving callbacks from the stock feed server.

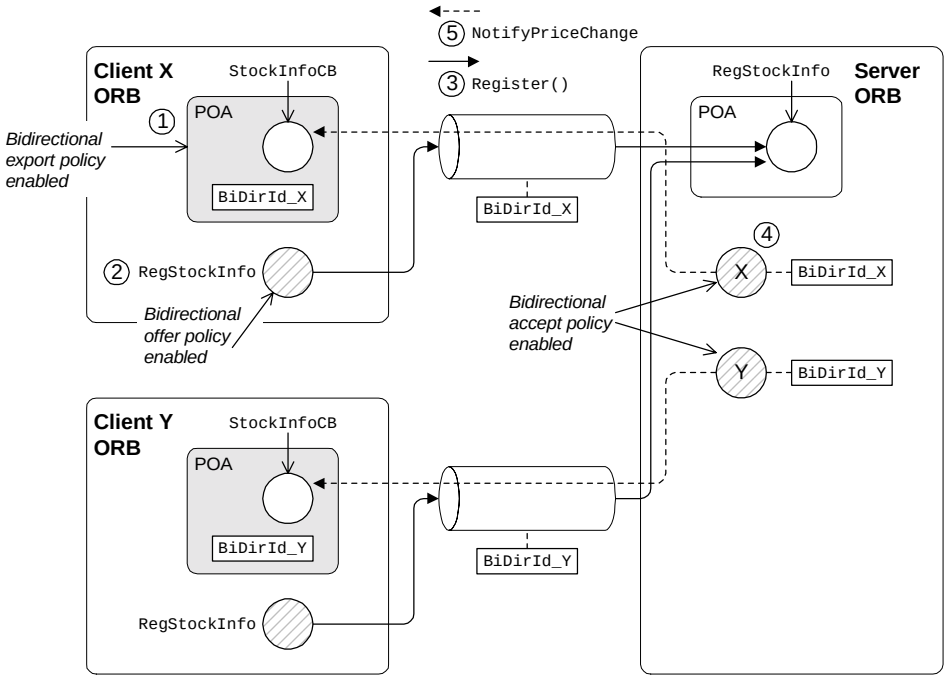


Figure 46: *Basic Bidirectional GIOP Scenario—Stock Feed*

Steps to establish a callback

Figure 46 shows the steps that occur to establish a stock feed callback, as follows:

Stage	Description
1	The client creates a POA instance, which has the `BidirectionalExportPolicy` enabled, and activates a `StockInfoCB` CORBA object, which is responsible for receiving callbacks. For the purposes of bidirectional GIOP, the POA is identified by the ID, `BiDirId_X`.
2	The client instantiates a `RegStockInfo` object reference, with the `BidirectionalOfferPolicy` enabled (the `RegStockInfo` object reference might have been retrieved from the naming service or from a stringified IOR).
3	The client invokes the `Register()` operation on the `RegStockInfo` object. A couple of things happen at this point: • The request message for the `Register` operation includes the `BiDirId_X` ID in a service context. This signals that the connection is offering to receive callbacks to the POA identified by `BiDirId_X`. • The `Register()` operation's argument is a reference to the `StockInfoCB` object, which will be used to accept callbacks from the server. The `StockInfoCB` object reference also has the `BiDirId_X` ID embedded in it.
4	If the `BidirectionalAcceptPolicy` policy is not already enabled at the level of the current ORB or the current thread, the server can enable this policy at the object level after receiving the `StockInfoCB` object reference (creating a new accept-enabled copy of the object reference).
5	Some time later, the server makes a callback on the client, calling the `NotifyPriceChange()` operation on the `StockInfoCB` object reference. Because the bidirectional accept policy is enabled on the object reference, Orbix checks to see whether it can re-use an existing incoming connection for the callback. By matching the GIOP `BiDirId` in the object reference to the GIOP `BiDirId` offered by a connection, Orbix finds a connection that it can re-use in bidirectional mode.

Setting the Export Policy

Overview

This subsection shows you how to set the `BiDirPolicy::BidirectionalExportPolicy` policy on a POA instance. This POA instance can then be used to activate CORBA objects that are intended to receive callbacks through a bidirectional GIOP connection.

Policy granularity

In this example, the `BiDirPolicy::BidirectionalExportPolicy` policy is set at *POA granularity*, which is the finest level of granularity for this policy.

Java example

Example 108 is a Java example that shows how to create a POA instance with the `BidirectionalExportPolicy` policy enabled. This POA instance is used on the client side to activate client callback objects.

Call the `org.omg.CORBA.ORB.create_policy()` method to create a `BidirectionalExportPolicy` object and then include this policy in the list of policies passed to the `org.omg.PortableServer.POA.create_POA()` method.

Example 108: *Java Setting the BidirectionalExportPolicy Policy*

```
// Java
// create callback POA with the effective
// BidirectionalExportPolicy set to ALLOW in order to allow an
// appropriate BiDirId be published in the callback reference
//
POA callback_poa = null;
try {
  ...
  System.out.println("creating the callback POA");
  Any export_value = orb.create_any();
  BidirectionalPolicyValueHelper.insert(
    export_value,
    ALLOW.value
  );
  Policy [] poa_policies = {
    orb.create_policy(BI_DIR_EXPORT_POLICY_TYPE.value,
    export_value)
  };
  callback_poa = root_poa.create_POA("callback",
                                     root_poa.the_POAManager(),
```

Example 108:*Java Setting the BidirectionalExportPolicy Policy*

```
                                        poa_policies);
}
catch ( /* ... */ )
{
  // Error handling ...
  // (Not shown)
}
```

Setting the Offer Policy

Overview

This subsection shows you how to set the `BiDirPolicy::BidirectionalOfferPolicy` policy on an object reference. After invoking an operation for the first time, the connection used by the object reference becomes available for bidirectional GIOP use. It does not matter whether the object reference opens a new connection or re-uses an existing connection.

For example, if an offer-enabled object reference re-uses an existing outgoing uni-directional connection, that connection becomes available for bidirectional use after the first invocation on the offer-enabled object reference.

Note: It might not be necessary to invoke an operation explicitly to make a connection available for bidirectional use. Sometimes operations are invoked implicitly—as, for example, when the `narrow()` function implicitly forces a remote `_is_a()` invocation.

Policy granularity

In this example, the `BiDirPolicy::BidirectionalOfferPolicy` policy is set at *object granularity*, which is the finest level of granularity for this policy.

Java example

Example 109 is a Java example that shows how to create a `RegStockInfo` object reference with the `BidirectionalOfferPolicy` policy enabled. This `RegStockInfo` object reference is used on the client side to connect to a `RegStockInfo` CORBA object on the server side.

Call the `org.omg.CORBA.ORB.create_policy()` method to create a `BidirectionalOfferPolicy` object and then include this policy in the list of policies passed to the `com.iona.corba.util.ObjectHelper.set_policy_overrides()` method.

Example 109: *Java Setting the BidirectionalOfferPolicy Policy*

```
// Java
// destringify RegStockInfo IOR and override the effective
// policies with the BidirectionalOfferPolicy set to ALLOW in
// order to allow a birectional offer be made with invocations on
```

Example 109: *Java Setting the BidirectionalOfferPolicy Policy*

```
// this reference - note the policy is overridden on the
   reference
// to be invoked by the client, not on the callback reference
//
RegStockInfo stock_registry = null;
try
{
  RandomAccessFile FileStream = new
   RandomAccessFile(Server.IOR_FILE, "r");
  String ior = FileStream.readLine();
  FileStream.close();
  org.omg.CORBA.Object obj = orb.string_to_object(ior);

  Any allow_value = orb.create_any();
  BidirectionalPolicyValueHelper.insert(allow_value,
   ALLOW.value);
  Policy [] policies = {
    orb.create_policy(BI_DIR_OFFER_POLICY_TYPE.value,
   allow_value)
  };
  stock_registry =
      RegStockInfoHelper.narrow(
        com.iona.corba.util.ObjectHelper.set_policy_overrides(
            obj,
            policies,
            SetOverrideType.ADD_OVERRIDE
        )
      );
}
catch( /* ... */ )
{
  // Error handling ...
  // (Not shown)
}
```

Setting the Accept Policy

Overview

This subsection shows you how to set the `BiDirPolicy::BidirectionalAcceptPolicy` policy on an object reference. In order to use an object reference on the server side as a bidirectional callback, the following prerequisites must be satisfied:

- The object reference is a proper *callback object reference*. For example, in Orbix 6.x a callback object reference has a `BiDirId` embedded in its IOR.
- The `BiDirPolicy::BidirectionalAcceptPolicy` policy must be enabled for the object reference.

When both of these prerequisites are satisfied, an operation invocation made on the callback object reference causes Orbix to attempt re-use an incoming connection in a bidirectional mode. An incoming connection is only considered for bidirectional use, if it offers the same `BiDirId` that appears in the callback object reference's IOR and the connection is compatible with the policies effective for the callback invocation.

Policy granularity

In this example, the `BiDirPolicy::BidirectionalAcceptPolicy` policy is set at *object granularity*, which is the finest level of granularity for this policy.

Java example

Example 110 is a Java example that shows how to create a `StockInfoCB` callback object reference with the `BidirectionalAcceptPolicy` policy enabled. This `StockInfoCB` callback object reference is used on the *server side* to connect to a `StockInfoCB` callback object on the *client side*.

Example 110: *Java Setting the BidirectionalAcceptPolicy Policy*

```
// Java
public void Register(StockInfoCB callback)
{
  System.out.println(
    "registration of interest in stock price changes"
  );

  // To accept the client's bidirectional offer, override
```

Example 110: *Java Setting the BidirectionalAcceptPolicy Policy*

```
  // the effective policies on the callback reference with the
  // BidirectionalAcceptPolicy set to ALLOW - similarly the
  // BidirectionalGen3AcceptPolicy is overridden to allow
  // bidirectional invocations on callback references registered
  // by gen3 clients
  //
  StockInfoCB accept_callback = null;
  try
  {
    Any allow_value = m_orb.create_any();
    BidirectionalPolicyValueHelper.insert(
        allow_value,
        ALLOW.value
    );
    Policy [] policies = {
1     m_orb.create_policy(
        BI_DIR_ACCEPT_POLICY_TYPE.value, allow_value
      ),
2     m_orb.create_policy(
        BI_DIR_GEN3_ACCEPT_POLICY_ID.value, allow_value
      )
    };
    org.omg.CORBA.Object o =
3       com.iona.corba.util.ObjectHelper.set_policy_overrides(
          callback,
          policies,
          SetOverrideType.ADD_OVERRIDE
    );
    accept_callback = StockInfoCBHelper.narrow(o);

    // add callback to list
    //
4   m_callbacks.add(accept_callback);
  }
  catch( /* ... */ )
  {
    // Error handling ...
    // (Not shown)
  }
  ...
}
```

The preceding Java code extract can be explained as follows:

1. This line calls the `org.omg.CORBA.ORB.create_policy()` method to create a `com.iona.BiDirPolicy.BidirectionalAcceptPolicy` object.
2. This line calls the `org.omg.CORBA.ORB.create_policy()` method to create a `com.iona.IT_BiDirPolicy.BidirectionalGen3AcceptPolicy` object. This proprietary policy allows you to accept bidirectional connections from Orbix 3 legacy clients. See "Interoperability with Orbix Generation 3" on page 595.
3. This line calls the `com.iona.corba.util.ObjectHelper.set_policy_overrides()` method to create a new object reference with the `BidirectionalAcceptPolicy` and `BidirectionalGen3AcceptPolicy` policies enabled.
4. The stock feed demonstration adds the callback object reference (with accept policies enabled) to its list of `StockInfoCB` object references.

Advanced BiDir Scenario

Overview

Figure 47 gives an overview of an advanced bidirectional scenario, where a client application establishes two separate connections to a server application. In this scenario, the server has to figure out which connection to use for the callback.

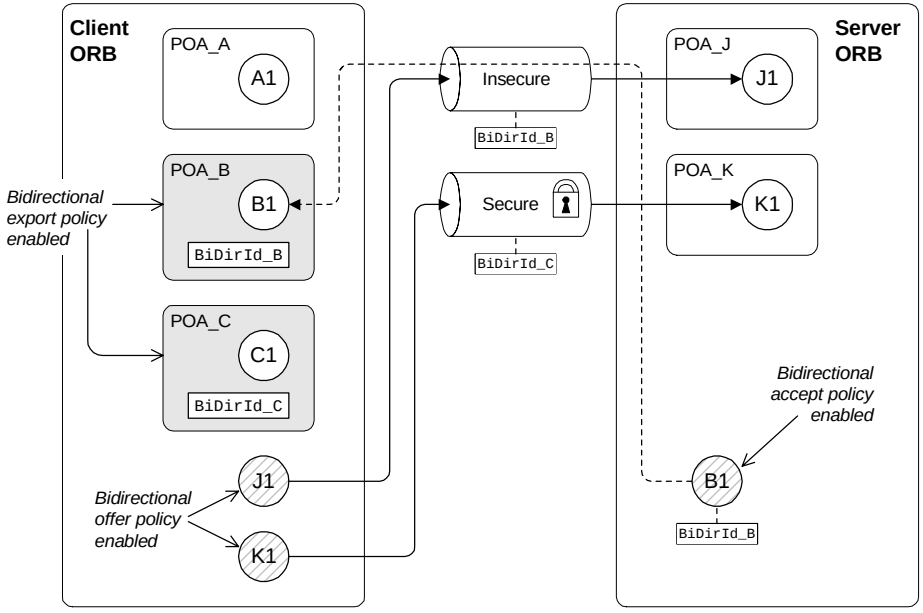


Figure 47: *Advanced Bidirectional GIOP Scenario*

Multiple endpoints

The main difference between this advanced bidirectional scenario, Figure 47, and the basic bidirectional scenario, Figure 46 on page 583, is that the advanced scenario features multiple endpoints, as follows:

- *Server-side endpoints*—`POA_J` and `POA_K`. The `POA_J` endpoint has its policies set so that it accepts insecure connections from clients. The `POA_K` endpoint has its policies set so that it requires secure connections from clients.
- *Client-side endpoints*—`POA_A`, `POA_B` and `POA_C`, of which only `POA_B` and `POA_C` can accept callbacks (their `BidirectionalExportPolicy` is set to `ALLOW`). `POA_B` is configured to accept only insecure callbacks. `POA_C` is configured to accept only secure callbacks.

Multiple connections

Because of the different security policies required by `POA_J` and `POA_K` in Figure 47, it is possible for one client application to establish multiple connections to the same server. For example, the client might establish an insecure connection to object `J1` in `POA_J`, and a secure connection to object `K1` in `POA_K`.

Bidirectional offer phase

The offer phase occurs whenever the client opens a connection to the server. In Figure 47, two offers are made:

- *Connection to the object,* `J1`—an *insecure* connection is made to the `POA_J` endpoint, which activates object `J1`. In the first request message over this connection, the client includes a `GIOP::BI_DIR_GIOP_OFFER` service context containing a list of the client endpoints that support insecure callbacks: that is, `BiDirId_B`.
- *Connection to the object,* `K1`—a *secure* connection is made to the `POA_K` endpoint, which activates object `K1`. Similarly to the first connection, the client includes a `GIOP::BI_DIR_GIOP_OFFER` service context containing a list of the client endpoints that support secure callbacks: that is, `BiDirId_C`.

Exporting a callback object

In Example 47 on page 592, the client exports a callback reference, `B1`, to the server. Because `POA_B` has its `BiDirExportPolicy` set to `ALLOW`, the IOR for `B1` includes a `GIOP::TAG_BI_DIR_GIOP` IOR component, which embeds the `BiDirId_B` bidirectional ID.

The presence of the `TAG_BI_DIR_GIOP` IOR component indicates to the server that the object, `B1`, supports bidirectional GIOP and the ID, `BiDirId_B`, identifies the associated endpoint on the client side.

Bidirectional accept phase

The accept phase occurs when the first operation invocation is made on the object reference, `B1`, on the server side. When the first operation is invoked on `B1`, the ORB recognizes that `B1` can use bidirectional GIOP, because the following conditions hold:

- The `BiDirAcceptPolicy` is set to `ALLOW` on the `B1` object reference, and
- The IOR for `B1` includes a `TAG_BI_DIR_GIOP` IOR component.

The ORB then extracts the `BiDirId_B` ID from `B1`'s IOR and compares this bidirectional ID with the offers from existing client connections. Because the insecure connection offers bidirectional GIOP for the `BiDirId_B` endpoint, the `B1` object reference attempts to re-use this connection for the callback. At this point, Orbix automatically compares the callback invocation policies with the attributes of the offered connection. Only if the policies are compatible will Orbix re-use the existing insecure connection for bidirectional GIOP.

Interoperability with Orbix Generation 3

Overview

Orbix 6.1 is designed to interoperate with Orbix 3 (Generation 3) clients. Figure 48 shows an example of the stock feed demonstration where one of the clients receiving callbacks is an Orbix 3 client.

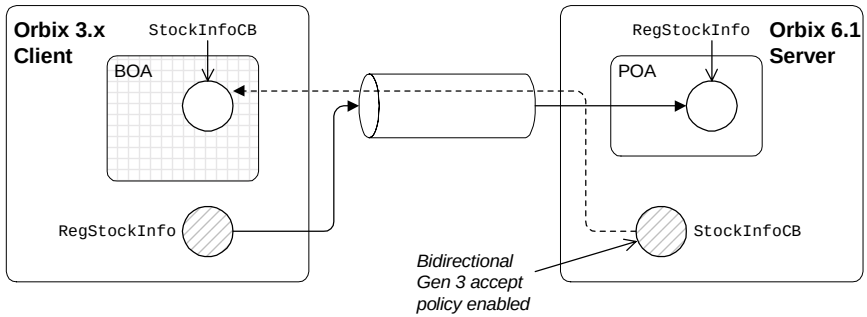


Figure 48: *Orbix 3 Client Receiving a Callback from an Orbix 6.1 Server*

Configuring an Orbix 6.1 server for Gen 3 interoperability

To configure an Orbix 6.1 server to interoperate bidirectionally with Orbix Generation 3 clients, you must include the appropriate `BiDir_Gen3` entry in the server's configured `binding:client_binding_list`. For example,

```
binding:client_binding_list = ["OTS+BiDir_GIOP", "BiDir_GIOP",
   "BiDir_Gen3", "OTS+GIOP+IIOP", "GIOP+IIOP", ...];
```

Setting the BiDir Gen 3 accept policy

To enable an Orbix 3 callback object reference to re-use an existing incoming connection on the server side, you must set the `IT_BiDirPolicy::BidirectionalGen3AcceptPolicy` on the callback object reference.

For Java example code, see Example 110 on page 589.

Asymmetry of Gen 3 bidirectional support

Orbix 6.1 support for Orbix 3 bidirectional GIOP is asymmetric. An Orbix 6.1 server can invoke on a Orbix 3 callback reference using bidirectional GIOP. However, an Orbix 6.1 client can not produce a callback reference that an Orbix 3 server could invoke on using bidirectional GIOP.

Limitations of Gen 3 bidirectional GIOP

Orbix 3 bidirectional GIOP is also subject to the following limitations:

- An Orbix 3 callback reference *must* be passed as a request parameter over the actual connection to be used for bidirectional invocations; whereas an Orbix 6.x bidirectional-enabled callback reference can be passed in any way to the server (for example, through the naming service or by stringifying to a shared file).
- The bidirectional offer implicit in an Orbix 3 callback reference is limited to the lifetime of the connection over which the callback reference is received by the server. Hence, further bidirectional invocations could not be made if, for example, the connection is reaped by the Orbix automatic connection management (ACM) and then re-established.

CHAPTER 21

Locating Objects with corbaloc

Corbaloc URLs enable you to specify the location of a CORBA service in a relatively simple format. Before using a corbaloc URL on the client side, you would normally register a simplified key for the CORBA object. Key registration can be done either using the itadmin named_key or by programming.

In this chapter

This chapter discusses the following topics:

corbaloc URL Format

Overview

The purpose of a corbaloc URL is to specify the location of a CORBA object in a human-readable format with the minimum amount of information necessary. For example, here is a typical example of a corbaloc URL:

```
corbaloc:iiop:1.2@LocatorHost:3075/NameService
```

Contrast this with a typical example of a stringified IOR:

```
IOR:010000003200000049444c3a696f6e612e636f6d2f49545f4f54535f53657
    27669636541646d696e2f53657276696365541646d696e3a312e30000000010
    00000000000008a0000000101020008000000066626f6c74616e00030c00003
    f0000003a3e0232310f73696d706c652e6c6f636174696f6e11694f5453006
    f7473746d0061646d696e00175472616e73616374696f6e536572766963654
    1646d696e0002000000010000001800000001000000010001000000000000
    1010001000000090101000600000006000000010000002600
```

There is an important difference between these two representations of an object reference: whereas the stringified IOR contains essentially the complete state of an object reference (including IOR components), the corbaloc URL contains only the object's address. Hence, object references constructed with a corbaloc URL are initialized in a provisional state. When an operation is first invoked on the object reference, Orbix exploits the GIOP location forward mechanism to retrieve the missing object reference details.

Converting a corbaloc URL to an object reference

In Java, you can convert a `corbaloc` URL into an object reference using the `org.omg.CORBA.ORB.string_to_object()` method, which has the following signature:

```
// Java
org.omg.CORBA.Object string_to_object(java.lang.String str);
```

For code examples, see "Using the corbaloc URL in a Client" on page 609.

corbaloc URL formats

The following corbaloc URL formats are described here:

- Basic corbaloc URL.
- Multiple-address corbaloc URL.
- Secure corbaloc URL.

Basic corbaloc URL

A basic `corbaloc` URL has the following format:

```
corbaloc:[iiop]:[Version@]Host[:Port][/ObjectKey]
```

The components of the basic `corbaloc` URL can be described as follows:

`iiop`	*(Optional)* Specifies the transport protocol to be IIOP. If omitted, the protocol defaults to IIOP. Hence, `corbaloc:iiop:` and `corbaloc::` are equivalent.
Version	*(Optional)* Specifies the GIOP version supported by the server. The allowed values are `1.0`, `1.1` and `1.2`; if omitted, the default is `1.0`. Orbix supports GIOP 1.2.
Host	Specifies the server's hostname or IP address in dotted decimal format.
Port	*(Optional)* Specifies the IP port used to connect to the server. If omitted, the default is `2809`.
ObjectKey	*(Optional)* A key that identifies the CORBA object on the remote server. According to the OMG specification, this key is the same as the object key that would be embedded in an equivalent IOR. To facilitate ease-of-use, however, Orbix provides mechanisms to substitute a human-readable key for the original object key.

Multiple-address corbaloc URL

The multiple-address corbaloc URL has the following format:

```
corbaloc:[CommaSeparatedAddressList][/ObjectKey]
```

With this form of corbaloc URL, you can locate a service that runs on more than one host and port (or is available through multiple protocols).

Each address in the list has the same format as the middle part of the basic corbaloc URL. For example, given that the FooService object is available both on HostX and HostY, you could specify a multiple-address corbaloc URL for the service as follows:

```
corbaloc:iiop:1.2@HostX:3075,iiop:1.2@HostY:3075/FooService
```

This form of URL is useful for specifying backup services; Orbix tries each of the addresses in the order in which they appear until it makes a successful connection.

Secure corbaloc URL

A secure corbaloc URL has the following format:

```
corbaloc:it_iiops:[Version@]Host[:Port][/ObjectKey]
```

This differs from the basic corbaloc URL only in that the transport protocol is it_iiops, which selects the IIOP/TLS protocol instead of IIOP. The it_iiops protocol specifier is Orbix-specific.

Note: Some earlier versions of Orbix (C++ only) used iiops to specify the IIOP/TLS protocol. If you need to support interoperability with older versions of Orbix, you could use a multiple-address corbaloc URL to support both types of protocol specifier, it_iiops and iiops.

For example, to connect securely to the FooService object:

```
corbaloc:it_iiops:1.2@FooHost:3075,iiops:1.2@FooHost:3075/FooSer
   vice
```

Object keys

The object key appearing in a corbaloc URL can have one of the following values:

- *Object key from an IOR*—the CORBA specification defines a `corbaloc` object key to be the same as the object key embedded in an IOR, except that non-printable characters are substituted by URL escape sequences. Unfortunately, this form of object key is unwieldy, because object keys from IORs are usually defined in a binary format.
- *Named key*—a named key is a human-readable key that is registered with the locator service. The named key enables you to construct a human-readable `corbaloc` URL for *indirect persistent servers*.
- *Plain text key*—a plain text key is a human-readable key that is registered with the `plain_text_key` plug-in. The plain text key enables you to construct a human-readable `corbaloc` URL for *direct persistent servers*.

The named key and the plain text key are conceptually similar; they are both mechanisms for substituting a human-readable key in a `corbaloc` URL.

Indirect Persistence Case

Overview

The mechanism used to substitute human-readable keys in a corbaloc URL must be tailored to the characteristics of the server, which could be either indirect persistent or direct persistent.

In the case of an indirect persistent server, the task of substituting human-readable keys is performed by the locator service, which maintains a *named key registry* in the IMR for this purpose.

In this section

This section contains the following subsections:

Overview of the Indirect Persistence Case

Overview

An *indirect persistent server* is a server that has a POA initialized with the following POA policy values:

- `PortableServer::LifespanPolicy` value is `PERSISTENT`, and
- `IT_PortableServer::PersistenceModePolicy` value is `INDIRECT_PERSISTENCE` (the default).

The CORBA objects activated by this POA have the following qualities:

- *Persistence*—implies that the object reference for this object remains valid even after the server is stopped and restarted.
- *Indirect persistence*—implies that clients establish contact with the server through the locator. In practice, the POA embeds the locator's address in the object references it generates. This forces clients to contact the locator before connecting to the server.

Figure 49 shows an overview of how Orbix resolves a `corbaloc` URL with the help of the locator service in the indirect persistent case.

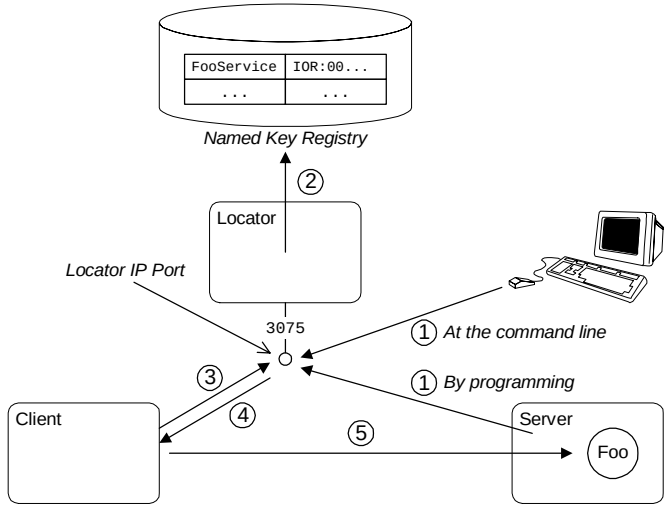


Figure 49: *Using corbaloc with the Locator-Based Named Key Registry*

Stages in registering and finding a named key

The stages involved in registering a named key and resolving a `corbaloc` URL constructed with that named key, as shown in Figure 49 on page 603, can be described as follows:

Stage	Description
1	There are two alternative ways to register a named key: • *At the command line*—use the `itadmin named_key create` command to associate a named key (for example, `FooService`) with a stringified IOR. • *By programming*—as the `Foo` service starts up, it contacts the locator to register the `FooService` named key against the `Foo` object reference.
2	The locator stores the `FooService` named key and object reference data in the named key registry, which is part of the implementation repository (IMR).
3	A client attempts to contact the server using the following URL: `corbaloc:iiop:1.2@LocatorHost:3075/FooService` Because the `corbaloc` URL contains the address of the locator, `LocatorHost:3075`, the client initially opens a connection to the locator service, sending either a GIOP LocateRequest message or a GIOP Request message.
4	The locator looks up the named key registry to find the object reference corresponding to the `FooService` key. The `Foo` object reference is then sent back to the client in a reply message (either a GIOP LocateReply message or a GIOP Reply message with LOCATION_FORWARD reply type).
5	Using the object reference data received from the locator, the client can now open a connection directly to the `Foo` server.

Registering a Named Key at the Command Line

Overview

To make a named key available for use in corbaloc URLs, the server must register the named key and its corresponding object reference in the named key registry. This subsection describes how to register a named key at the command line.

The itadmin named_key command

The `itadmin named_key` command supports a variety of subcommands for managing named keys in the implementation repository, as follows:

`named_key create`	Creates an association between a specified well-known object key and a specified object reference.
`named_key list`	Lists all well known object keys that are registered with the locator daemon.
`named_key remove`	Removes the specified object key from the location domain.
`named_key show`	Displays the object reference associated with the given key.

For full details of these commands, see the *Orbix Administrator's Guide*.

Creating a named key using itadmin named_key create

To create a named key using the `itadmin named_key create` command, perform the following steps:

Step	Action
1	Obtain a stringified IOR for the CORBA object that you want to register. You could obtain the IOR in one of the following ways: • If the server dumps the stringified IOR to a file or to the console window, you can copy it from there (the `org.omg.CORBA.ORB.object_to_string()` method generates stringified IORs). • If the object is already registered in the CORBA naming service, you can obtain the stringified IOR using the `itadmin ns resolve` *Name* command.
2	Register the stringified IOR from the preceding step, *String-IOR*, associating it with a named key, *NamedKey*, by entering the following command: `itadmin named_key create -key` *NamedKey String-IOR*

Registering a Named Key by Programming

Overview

This subsection describes the alternative approach to registering corbaloc URLs in the named key registry, which is by programming. A code example shows how a server contacts the locator service to register a named key.

Server example in Java

Example 111 shows how a server can register a named key, `FooService`, that identifies a given object reference, `FooObjectRef` (the object reference must have been generated from a CORBA object belonging to an indirect persistent POA).

Example 111: *Registering a Named Key with the Locator*

```
// Java
...
// Get the Locator
org.omg.CORBA.Object objref =
1       orb.resolve_initial_references("IT_Locator");
com.iona.corba.IT_Location.Locator locator =
       com.iona.corba.IT_Location.LocatorHelper.narrow(objref);

// Get the Named Key registry
objref = locator.resolve_service(
2            com.iona.corba.IT_NamedKey.NAMED_KEY_REGISTRY
        );
3  com.iona.corba.IT_NamedKey.NamedKeyRegistry registry =
     com.iona.corba.IT_NamedKey.NamedKeyRegistryHelper.narrow(
       objref
    );

// Add a key to the registry
try
{
4    registry.add_text_key("FooService", FooObjectRef);
}
catch
   (com.iona.corba.IT_NamedKey.NamedKeyRegistryPackage.EntryAlre
   adyExists  ex)
{
    // Error: ...
}
```

The preceding Java code example can be explained as follows:

1. The `IT_Locator` initial reference ID is used to obtain a reference to the `IT_Location::Locator` IDL interface. The `Locator` interface enables a server to communicate directly with the Orbix locator service (the `IT_Location` IDL module is defined in the *OrbixInstallDir*`/asp/`*Version*`/idl/orbix/location.idl` file).
2. The `resolve_service()` operation is called to return a reference to the named key registry. The `com.iona.corba.IT_NamedKey.NAMED_KEY_REGISTRY` is a string constant, which has the value `IT_NamedKey::NamedKeyRegistry`.
3. The `IT_NamedKey::NamedKeyRegistry` IDL interface defines operations to register named keys and manage the named key registry. See the *Java Programmer's Reference* for more details.
4. The `com.iona.corba.IT_NamedKey.NamedKeyRegistry.add_text_key()` method registers a new named key with the locator.

Using the corbaloc URL in a Client

Overview

The usual format for a corbaloc URL that references an indirect persistent server is as follows:

```
corbaloc:iiop:1.2@LocatorHost:LocatorPort/NamedKey
```

Because the server is indirect persistent, the URL embeds the locator's address, *LocatorHost*:*LocatorPort*, not the server's own address.

For example, given that the Orbix locator is running on host, `LocatorHost`, and port, `3075`, and the server registers a `Foo` object under the named key, `FooService`, you could access the `Foo` object with the following URL:

```
corbaloc:iiop:1.2@LocatorHost:3075/FooService
```

Client example in Java

Example 112 shows how to resolve a corbaloc URL for an object of `Foo` type, using the `org.omg.CORBA.ORB.string_to_object()` method.

Example 112: *Resolving a corbaloc URL*

```
// Java
try {
  java.lang.String corbalocURL =
   "corbaloc:iiop:1.2@LocatorHost:3075/FooService";

  org.omg.CORBA.Object objref =
   orb.string_to_object(corbalocURL);

  Foo fooObj= FooHelper.narrow(objref);
  if (CORBA::is_nil(fooObj)) {
      // Error: _narrow failed!
  }
}
catch (org.omg.CORBA.BAD_PARAM ex) {
    // Error: narrow failed!
}
catch (org.omg.CORBA.SystemException sysex) {
    // Error: general error
}
```

Direct Persistence Case

Overview

The mechanism used to substitute human-readable keys in a corbaloc URL must be tailored to the characteristics of the server, which could be either indirect persistent or direct persistent.

In the case of a direct persistent server, the task of substituting human-readable keys is performed by the `plain_text_key` plug-in, which holds a transient list of *plain text keys* for this purpose.

In this section

This section contains the following subsections:

Overview of the Direct Persistence Case

Overview

A *direct persistent server* is a server that has a POA initialized with the following POA policy values:

- `PortableServer::LifespanPolicy` value is `PERSISTENT`, and
- `IT_PortableServer::PersistenceModePolicy` value is `DIRECT_PERSISTENCE`.

The CORBA objects activated by this POA have the following qualities:

- *Persistence*—implies that the object reference for this object remains valid even after the server is stopped and restarted.
- *Direct persistence*—implies that clients establish contact with the server directly, bypassing the locator. Hence, the POA embeds the server's own address in the object references it generates.

Figure 50 shows an overview of how Orbix resolves a `corbaloc` URL using the `plain_text_key` plug-in in the direct persistent case.

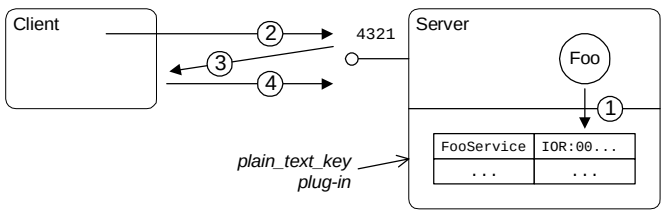


Figure 50: *Using corbaloc with the plain_text_key Plug-In*

Stages in registering and finding a plain text key

The stages involved in registering a plain text key and resolving a `corbaloc` URL constructed with that plain text key, as shown in Figure 50 on page 611, can be described as follows:

Stage	Description
1	As the `Foo` service starts up, it registers the `FooService` plain text key with the `plain_text_key` plug-in.
2	A client attempts to contact the server using the following URL: `corbaloc:iiop:1.2@FooHost:4321/FooService` Because the `corbaloc` URL contains the address of the `Foo` server, `FooHost:4321`, the client opens a connection directly to the server (sending either a GIOP LocateRequest message or a GIOP Request message).
3	The `plain_text_key` plug-in finds the object reference corresponding to the `FooService` key. The `Foo` object reference is then sent back to the client in a reply message (either a GIOP LocateReply message or a GIOP Reply message with LOCATION_FORWARD reply type).
4	Using the object reference data received in the previous step, the client now resends the GIOP Request message to the server.

Registering a Plain Text Key

Overview

To make a plain text key available for use in corbaloc URLs, the server must register the plain text key and its corresponding object reference with the `plain_text_key` plug-in.

Server example in Java

Example 113 shows how a server registers a plain text key, `FooService`, that identifies a given object reference, `FooObjectRef` (the object reference must have been generated from a CORBA object belonging to a direct persistent POA).

Example 113: *Registering a Plain Text Key*

```
// Java
// Try/Catch block not shown ...
org.omg.CORBA.Object objref =
   the_orb.resolve_initial_references(
1                                  "IT_PlainTextKeyForwarder"
                                );
com.iona.corba.IT_PlainTextKey.Forwarder forwarder =

   com.iona.corba.IT_PlainTextKey.ForwarderHelper.narrow(objref)
   ;

2  forwarder.add_plain_text_key(
               "FooService",
               FooObjectRef
          );
```

The preceding Java code can be explained as follows:

1. The `IT_PlainTextKeyForwarder` initial reference ID is used to obtain a reference to a `com.iona.corba.IT_PlainTextKey.Forwarder` object (the `IT_PlainTextKey` IDL module is defined in the *OrbixInstallDir*`/asp/`*Version*`/idl/orbix_pdk/plain_text_key.idl` file).
2. The `add_plain_text_key()` method adds a new plain text key to the list held by the `plain_text_key` plug-in.

Using the corbaloc URL in a Client

Overview

The usual format for a `corbaloc` URL that references a direct persistent server is as follows:

```
corbaloc:iiop:1.2@ServerHost:ServerPort/PlainTextKey
```

Because the server is direct persistent, the URL embeds the server's own address, *ServerHost*:*ServerPort*.

For example, given that the server is running on host, `FooHost`, and port, `4321`, and the server registers a `Foo` object under the plain text key, `FooService`, you could access the `Foo` object with the following URL:

```
corbaloc:iiop:1.2@FooHost:4321/FooService
```

Client example in Java

Example 114 shows how to resolve a `corbaloc` URL for an object of `Foo` type, using the `org.omg.CORBA.ORB.string_to_object()` method.

Example 114: *Resolving a corbaloc URL*

```
// Java
try {
  java.lang.String corbalocURL =
   "corbaloc:iiop:1.2@FooHost:4321/FooService";

  org.omg.CORBA.Object objref =
   orb.string_to_object(corbalocURL);

  Foo fooObj= FooHelper.narrow(objref);
  if (CORBA::is_nil(fooObj)) {
      // Error: _narrow failed!
  }
}
catch (org.omg.CORBA.BAD_PARAM ex) {
    // Error: narrow failed!
}
catch (org.omg.CORBA.SystemException sysex) {
    // Error: general error
}
```

CHAPTER 22

Configuring and Logging

Orbix has built-in configuration and logging mechanisms, which are used internally by the Orbix product. You have the option of using these configuration and logging mechanisms in your own applications.

In this chapter

This chapter discusses the following topics:

The Configuration Interface

The IT_Config::Configuration interface

The `Configuration` interface is defined as a local interface within the `IT_Config` module, as follows:

Example 115: *Definition of the IT_Config::Configuration IDL Interface*

```
# Orbix Configuration File
...
#pragma prefix "iona.com"

module IT_Config
{
    typedef sequence<string> ConfigList;
    ...
    exception TargetNotFound {};

    local interface Configuration
    {
        exception TypeMismatch {};

        boolean get_string(in string  name, out string value)
            raises (TypeMismatch);

        boolean get_list(in string name, out ConfigList value)
            raises (TypeMismatch);

        boolean get_boolean(in string   name, out boolean value)
            raises (TypeMismatch);

        boolean get_long(in string name, out long  value)
            raises (TypeMismatch);

        boolean get_double(in string  name, out double value)
            raises (TypeMismatch);
        ...
    };
    ...
};
...
```

The ConfigList type

The `IT_Config::ConfigList` type, which is defined as a sequence of strings, is used to hold the data returned from the `Configuration::get_list()` operation. The following configuration variable, `my_list_item`, is an example of a configuration entry that needs to be retrieved as a list, using `get_list()`.

```
# Orbix Configuration
my_list_item = ["first", "second", "third"];
```

Operations

The following operations of the `Configuration` interface are listed in Example 115 on page 616:

- `get_string()`—get the value of the `name` variable as a string type.
- `get_list()`—get the value of the `name` list variable as a list of strings, `IT_Config::ConfigList`.
- `get_boolean()`—get the value of the `name` variable as a CORBA boolean type.
- `get_long()`—get the value of the `name` variable as a CORBA long type.
- `get_double()`—get the value of the `name` variable as a CORBA double type.

Reference

For more details of the `Configuration` interface and the `IT_Config` module, see the `IT_Config` sections of the *CORBA Programmer's Reference*.

Configuring

Overview

Orbix has a flexible configuration system which enables an application to retrieve configuration data without needing to know anything about the actual source of the data. This section briefly describes Orbix configuration, covering the following topics:

- Generating configuration domains.
- Configuration sources.
- Sample configuration.
- Java accessing configuration settings.
- References.

Generating configuration domains

Configuration domains are generated by running the `itconfigure` tool.

Configuration sources

Orbix configuration data can come from one of the following sources:

- *Configuration file*—this is a file, *DomainName*`.cfg`, that stores configuration settings in a format that is easily readable and editable.
- *Configuration repository (CFR) service*—this is a service that stores configuration settings in a central database and is remotely accessible to CORBA applications. Note, that a minimal configuration handler file, *DomainName*`.cfg`, is also needed on hosts that use the CFR service in order to contact the CFR initially.

Sample configuration

Example 116 shows some sample configuration settings, of various types, that might be used to configure a `hello_world` plug-in.

Example 116: *Sample Configuration Settings*

```
# Orbix configuration file
plugin_example {
    plugin:hello_world:boolean_item = "true";
    plugin:hello_world:string_item  = "Hello World!";
    plugin:hello_world:long_item    = "4096";
    plugin:hello_world:double_item  = "3.14";
    plugin:hello_world:list_item    = ["first", "second",
   "third"];
};
```

Java accessing configuration settings

Example 117 shows how to access configuration settings in Java. There are two main steps in this code extract:

1. The application obtains an initial reference to an `com.iona.corba.IT_Config.Configuration` object
2. The application reads configuration data using the methods defined on the `IT_Config.Configuration` interface.

Example 117: *Java Accessing Configuration Settings*

```
// Java
...
import com.iona.corba.IT_Config.*;
import com.iona.corba.IT_Config.ConfigurationPackage.*;
...

private void load_config()
  {
    org.omg.CORBA.Object initial_reference = null;
    Configuration        config            = null;

    // 1. Obtain an initial reference to the configuration.
    //
    try {
      initial_reference = m_orb.resolve_initial_references(
                              "IT_Configuration"
                          );
      config = ConfigurationHelper.narrow(initial_reference);
```

Example 117:*Java Accessing Configuration Settings*

```
    }
    catch(org.omg.CORBA.ORBPackage.InvalidName in) {
      // Handle InvalidName error...
    }
    catch(java.lang.Exception e) {
      // Handle generic error...
    }

    // 2. Read some configuration variables.
    //
    try  {
      org.omg.CORBA.BooleanHolder tmp_bool =
          new org.omg.CORBA.BooleanHolder();
      config.get_boolean(
          "plugin:hello_world:boolean_item", tmp_bool
      );
      m_boolean_item = tmp_bool.value;

      org.omg.CORBA.StringHolder tmp_string =
          new org.omg.CORBA.StringHolder();
      config.get_string(
          "plugin:hello_world:string_item", tmp_string
      );
      m_string_item = tmp_string.value;

      org.omg.CORBA.IntHolder tmp_long =
          new org.omg.CORBA.IntHolder();
      config.get_long("plugin:hello_world:long_item", tmp_long);
      m_long_item = tmp_long.value;

      org.omg.CORBA.DoubleHolder tmp_double =
          new org.omg.CORBA.DoubleHolder();
      config.get_double(
          "plugin:hello_world:double_item", tmp_double
      );
      m_double_item = tmp_double.value;

      com.iona.corba.IT_Config.ConfigListHolder tmp_list =
          new com.iona.corba.IT_Config.ConfigListHolder();
      config.get_list(
          "plugin:hello_world:list_item", tmp_list
      );
      m_list_item = tmp_list.value;
}
```

The last item read is a configuration list. The `m_list_item` variable is an array of strings, which is of `java.lang.String[]` type.

References

The following references can provide you with more information about Orbix configuration:

- The documentation of the `IT_Config::Configuration` interface in the *CORBA Programmer's Reference*.

Logging

Overview

Logging provides administrators and system operators with information about a production system, reporting information such as significant system events, warnings of anomalous conditions, and detailed information about error conditions. Its primary goal is to provide administrators with the information needed to detect diagnose and resolve problems in a production system.

Logging event

An Orbix logging event has the following structure:

- Logging subsystem.
- Event ID.
- Event priority.
- Message.

Logging subsystem

A *logging subsystem*, identified by a *subsystem ID*, provides a convenient way of grouping together related logging events and messages. The subsystem ID is useful when it comes to filtering log events, because you can use it to specify logging options on a per-subsystem basis.

Typically, a unique logging subsystem is defined for each plug-in. For example, the lease plug-in defines its own logging subsystem, `IT_LEASE`, as shown in Example 118 on page 624.

Event ID

An *event ID* is a constant, of `IT_Logging::EventId` type, that identifies a particular type of event.

Before you can use logging in your own plug-in code, you must define a collection of custom event IDs in IDL. See Example 118 on page 624 for an example of how this is done for the leasing plug-in.

Event priority

Every event that is generated must have a priority assigned to it.

In Java, you can use one of the following constants (of `short` Java type) to assign priority to an event:

```
com.iona.corba.IT_Logging.LOG_INFO.value
com.iona.corba.IT_Logging.LOG_WARNING.value
com.iona.corba.IT_Logging.LOG_ERROR.value
com.iona.corba.IT_Logging.LOG_FATAL_ERROR.value
```

Message

A log message is a string, which might include some embedded parameters.

Local log stream

The *local log stream* reports events either to a local file or to standard error. You can enable the local log stream by including `local_log_stream` in your list of `orb_plugins`, as follows:

```
# Orbix configuration file
plugin_example {
    orb_plugins = ["local_log_stream", "iiop_profile", "giop",
   "iiop", "hello_world"];
    ...
};
```

For more details about how to configure a local log stream, see the *CORBA Administrator's Guide*.

System log stream

The *system log stream* reports events to the host's system log. You can enable the system log stream by including `system_log_stream` in your list of `orb_plugins`, as follows:

```
# Orbix configuration file
plugin_example {
    orb_plugins = ["system_log_stream", "iiop_profile", "giop",
   "iiop", "hello_world"];
    ...
};
```

For more details about how to configure a system log stream, see the *CORBA Administrator's Guide*.

Defining a subsystem ID and event IDs

Before you can use logging with your plug-in, you must define a logging subsystem ID and a set of event IDs in IDL.

For example, the IDL in Example 118 shows the subsystem ID and event IDs defined for the lease plug-in.

Example 118: *Example Subsystem ID and Event ID Definitions*

```
#include <orbix/logging.idl>

module IT_Lease_Logging
{
    const IT_Logging::SubsystemId SUBSYSTEM = "IT_LEASE";

    // Errors (1+)
    //
    const IT_Logging::EventId NAMING_SERVICE_UNREACHABLE     = 1;
    const IT_Logging::EventId REAPER_THREAD_FAILURE          = 2;
    const IT_Logging::EventId RENEWAL_THREAD_FAILURE         = 3;
    const IT_Logging::EventId CALLBACK_FAILURE               = 4;
    const IT_Logging::EventId INVALID_LEASE_AGENT_REFERENCE  = 5;
    const IT_Logging::EventId LEASE_AGENT_NOT_FOUND          = 6;
    const IT_Logging::EventId LEASE_ACQUISITION_FAILURE      = 7;

    // Warnings (100+)
    //
    const IT_Logging::EventId CLIENT_LEASE_RELEASE_FAILURE = 100;
    const IT_Logging::EventId SERVER_LEASE_WITHDRAW_FAILURE= 101;
    const IT_Logging::EventId DEFAULT_REAP_TIME_USED       = 102;
    const IT_Logging::EventId DEFAULT_PING_TIME_USED       = 103;
    const IT_Logging::EventId PING_TIME_ALTERED            = 104;
    const IT_Logging::EventId LEASE_EXPIRED_PREMATURELY    = 105;

    // Informational messages (200+)
    //
    const IT_Logging::EventId CLIENT_LEASES_UPDATED        = 200;
    const IT_Logging::EventId SERVER_LEASES_UPDATED        = 201;
    const IT_Logging::EventId CONFIGURATION_DUMP           = 202;
    const IT_Logging::EventId SERVER_LEASE_REAPER_CHECK    = 203;
    const IT_Logging::EventId LEASE_EXPIRATION             = 204;
    const IT_Logging::EventId LEASE_ADVERTISED_OK          = 205;
    const IT_Logging::EventId RENEWAL_NOT_NEEDED_YET       = 206;
    const IT_Logging::EventId RENEWING_LEASE               = 207;
};
```

Java logging messages

Example 119 shows an extract from the lease plug-in code, which shows how to obtain a reference to an event log and send messages to the event log.

Example 119: *Java Example of Logging Messages*

```
// Java
...
import com.iona.corba.IT_Logging.*;
import com.iona.corba.IT_Lease_Logging.*;

  LeasePerORBState(ORB orb)
    throws INTERNAL
  {
    org.omg.CORBA.Object initial_reference = null;
    m_orb = orb;

    // Get the Event Log
    try {
1     initial_reference = m_orb.resolve_initial_references(
                              "IT_EventLog"
                          );
    }
    catch(org.omg.CORBA.ORBPackage.InvalidName in) {
      throw new INTERNAL();
    }
2   m_event_log = EventLogHelper.narrow(initial_reference);
    ...

    // Example log message:
    // The leasing plug-in logs this message if it fails to
    // connect to the CORBA Naming Service.
    //
3   m_event_log.report_message(
     SUBSYSTEM.value,
     NAMING_SERVICE_UNREACHABLE.value,
     LOG_ERROR.value,
     LeaseEventMessages.IT_LEASE_NAMING_SERVICE_UNREACHABLE_MSG,
     new org.omg.CORBA.Any[0]
    );
    ...
  }
```

The preceding Java logging example can be explained as follows:

1. This line obtains an initial reference to the `com.iona.corba.IT_Logging.EventLog` object, by calling `resolve_initial_references()` with the `IT_EventLog` initial object ID string.
2. Narrow the initial reference to `m_event_log`, which has been declared elsewhere to be of `com.iona.corba.IT_Logging.EventLog` type.
3. The `report_message()` method sends events/messages to the event log. The method takes the following parameters:
   - A subsystem ID, of `java.lang.String` type.
   - An event ID, of `int` Java type.
   - An event priority, of `short` Java type.
   - A message string, of `java.lang.String` type.
   - An array of message parameters, of `org.omg.CORBA.Any[]` type. These are parameters that can optionally be embedded in the message string. The message string references the parameters using the symbols `%0`, `%1`, `%2`, and so on.

References

The following resources are available on the subject of Orbix logging:

- The documentation of the `IT_Logging` module in the *CORBA Programmer's Reference*.

Orbix IDL Compiler Options

The IDL compiler compiles the contents of an IDL module into header and source files for client and server processes, in the specified implementation language. You invoke the `idl` compiler with the following command syntax:

```
idl -plugin[...] [-switch]... idlModule
```

Note: You must specify at least one plug-in switch, such as `-poa` or `-base`, unless you modify the IDL configuration file to set `IsDefault` for one or more plug-ins to Yes. (see page 634). As distributed, the configuration file sets `IsDefault` for all plug-ins to No.

Command Line Switches

You can qualify the `idl` command with one or more of the following switches. Multiple switches are colon-delimited.

Switch	Description
`-Dname[:value]`	Defines the preprocessor's name.
`-E`	Runs preprocessor only, prints on `stdout`.
`-Idir`	Includes *`dir`* in search path for preprocessor.
`-R[-v]`	Populates the interface repository (IFR). The -v modifier specifies verbose mode.
`-Uname`	Undefines name for preprocessor.
`-V`	Prints version information and exits.
`-u`	Prints usage message and exits.
`-w`	Suppresses warning messages.
`-plugin [:-modifier]...`	Specifies to load the IDL plug-in specified by *`plug-in`* to generate code that is specific to a language or ART plug-in. You must specify at least one plug-in to the idl compiler Use one of these values for *`plug-in`*: • `base`: Generate C++ header and stub code. • `jbase`: Generate Java stub code • `poa`: Generate POA code for C++ servers. • `poa`: Generate POA code for Java servers. • `psdl`: Generate C++ code that maps to abstract PSDL constructs. • `pss_r`: Generate C++ code that maps concrete PSDL constructs to relational and relational-like database back-end drivers. Each *`plug-in`* switch can be qualified with one or more colon-delimited modifiers.

Plug-in Switch Modifiers

The following tables describe the modifiers that you can supply to plug-in switches such as `-base` or `-poa`.

Table 22: *Modifiers for all C++ plug-in switches*

Modifier	Description

`-d[decl-spec]`	Creates NT declspecs for `dllexport` and `dllimport`. If you omit *`decl-spec`*, `idl` uses the stripped IDL module's name.
	For example, the following command:
	`idl -dIT_ART_API foo.idl`
	yields this code:
	`#if !defined(IT_ART_API)`
	`#if defined(IT_ART_API_EXPORT)`
	`#define IT_ART_API IT_DECLSPEC_EXPORT`
	`#else`
	`#define IT_ART_API IT_DECLSPEC_IMPORT`
	`#endif`
	`#endif`
	If you compile and link a DLL with the `idl`-generated code within it, `IT_ART_API_EXPORT` must be a defined preprocessor symbol so that `IT_ART_API` is set to `dllexport`. All methods and variables in the generated code can be exported from the DLL and used by other applications. If `IT_ART_API_EXPORT` is not defined as a preprocessor symbol, `IT_ART_API` is set to `dllimport`; methods and variables that are defined in the generated code are imported from a DLL.

`-ipath-prefix`	Prepends *`path-prefix`* to generated `include` statements. For example, if the IDL file contains the following statement:
	`#include "foo.idl"`
	`idl` generates this statement in the header file:
	`#include path-prefix/foo.hh`

Table 22: *Modifiers for all C++ plug-in switches*

Modifier	Description

`-h[suffix.]ext`	Sets header file extensions. The default setting is `.hh`.
	For example, the following command:
	`idl -base:-hh foo.idl`
	yields a header file with this name:
	`foo.h`
	If the argument embeds a period (.), the string to the left of the period is appended to the IDL file name; the string to the right of the period specifies the file extension. For example, the following command:
	`idl -base:-h_client.h foo.idl`
	yields the following header file name:
	`foo_client.h`
	If you use the `-h` to modify the `-base` switch, also use `-b` to modify the `-poa` switch (see Table 25).

`-Oh`*`path`*	Sets the output directory for header files.
`-Oc`*`path`*	Sets the output directory for client stub (`.cxx`) files.
`-xAMICallbacks`	Generates stub code that enables asynchronous method invocations (AMI).

Table 23: *Modifier for -base, -psdl, and -pss_r plug-in switches*

Modifier	Description

`-c[suffix.]ext`	Specifies the format for stub file names. The default name is *`idl-name`*`.cxx`.
	For example, the following command:
	`idl -base:-cc foo.idl`
	yields a server skeleton file with this name:
	`foo.c`
	If the argument embeds a period (.), the string to the left of the period is appended to the IDL file name; the string to the right of the period specifies the file extension. For example, the following command:
	`idl -base:-c_client.c foo.idl`
	yields the following stub file name:
	`foo_client.c`

Table 23: *Modifier for -base, -psdl, and -pss_r plug-in switches*

Modifier	Description
`-xOBV`	Generates object-by-value default `valuetype` implementations in files.

Table 24: *Modifiers for -jbase and -jpoa switches*

Modifier	Description
`-P`*`package`*	Use *package* as the root scope to package all unspecified modules. By default, all Java output is packaged in the IDL module names.
`-P`*`module=package`*	Use *package* as the root scope for the specified module.
`-O`*`dir`*	Output all java code to *dir*. The default is the current directory.
`-Gdsi` `-Gstream`	Output DSI or stream-based code. The default is `stream`.
`-Mreflect` `-Mcascade`	Specifies the POA dispatch model to use either reflection or cascading `if-then-else` statements. The default is `reflect`.
`-J1.1` `-J1.2`	Specifies the JDK version. The default is 1.2.
`-VTRUE` `-VFALSE`	Generate native implementation for valuetypes. The default is `FALSE`.
`-FTRUE` `-FFALSE`	Generate factory implementation for valuetypes. The default is `FALSE`.
`-ETRUE` `-EFALSE`	Initialize the string fields of structures and exceptions to the empty string. The default is `FALSE`, meaning that string fields are initialized to `null`.

Table 25: *Modifiers for -poa switch*

Modifier	Description

`-s[suffix.]ext`	Specifies the skeleton file name. The default name is *idl-name*`S.cxx` for skeleton files.
	For example, the following command:
	`idl -poa:-sc foo.idl`
	yields a server skeleton file with this name:
	`fooS.c`
	If the argument embeds a period (.), the string to the left of the period is appended to the IDL file name; the string to the right of the period specifies the file extension. For example, the following command:
	`idl -poa:-s_server.h foo.idl`
	yields the following skeleton file name:
	`foo_server.c`

`-b[suffix.]ext`	Specifies the format of the header file names in generated `#include` statements. Use this modifier if you also use the -h modifier with the `-base` plug-in switch.
	For example, if you specify a `.h` extension for `-base`-generated header files, specify the same extension in `-poa`-generated `#include` statements, as in the following commands:
	`idl -base:-hh foo.idl`
	`idl -poa:-bh foo.idl`
	These commands generate header file `foo.h`, and include in skeleton file `fooS.cxx` a header file of the same name:
	`#include "foo.h"`
	If the argument embeds a period (.), the string to the left of the period is appended to the IDL file name; the string to the right of the period specifies the file extension. For example, the following command:
	`idl -poa:-b_client.h foo.idl`
	yields in the generated skeleton file the following `#include` statement:
	`#include "foo_client.h"`

Table 25: *Modifiers for -poa switch*

Modifier	Description

`-m`*`incl-mask`*	`#include` statements with file names that match *`mask`* are ignored in the generated skeleton header file. This lets the code generator ignore files that it does not need. For example, the following switch:
	`-momg/orb`
	directs the `idl` compiler to ignore this `#include` statement in the IDL/PSDL:
	`#include <omg/orb.idl>`

`-p`*`multiple`*	Sets the dispatch table to be 2 to the power of *`multiple`*. The default value of *`multiple`* is 1. Larger dispatch tables can facilitate operation dispatching, but also increase code size and memory usage.
`-xTIE`	Generates POA TIE classes.

IDL Configuration File

The IDL configuration file defines valid `idl` plug-in switches such as `-base` and `-poa` and specifies how to execute them. For example, the default IDL configuration file defines the `base` and `poa` switches, the path to their respective libraries, and command line options to use for compiling C++ header and client stub code and POA code.

IDL configuration files have the following format:

Figure 51: *Configuration file format*

```
IDLPlugins = "plugin-type[, plugin-type].."

plugin-type
{
   Switch = switch-name;
   ShlibName = path;
   ShlibMajorVersion = version
   ISDefault = "{ YES | NO }";
   PresetOptions = "-plugin-modifier[, -plugin-modifier]..."

#  plugin-specific settings...
#  ...
}
```

plugin-type can be one of the following literals:

```
Java
POAJava
Cplusplus
POACxx
IFR
PSSDLCxx
PSSRCxx
```

The `idl` command can supply additional switch modifiers; these are appended to the switch modifiers that are defined in the configuration file. You can comment out any line by beginning it with the # character.

The distributed IDL configuration file looks like this:

Figure 52: *Distributed IDL configuration file*

```
# IDL Configuration File

# IDL_CPP_LOCATION configures the C-Preprocessor for the IDL
# Compiler
# It can be the fully qualified path with the executable name or
# just the executable name
#IDL_CPP_LOCATION = "%PRODUCT_BIN_DIR_PATH%/idl_cpp";
#IDL_CPP_ARGUMENTS = "";
#tmp_dir = "c:\temp";

IDLPlugins = "Java, POAJava, Cplusplus, POACxx, IFR, PSSDLCxx,
    PSSRCxx";

Cplusplus
{
        Switch = "base";
        ShlibName = "it_cxx_ibe";
        ShlibMajorVersion = "1";
        IsDefault = "NO";
        PresetOptions = "-t";

#       Header and StubExtension set the generated files
   extension
#       The Default is .cxx and .hh
#
#       StubExtension = "cxx";
#       HeaderExtension = "hh";
};
```

Figure 52: *Distributed IDL configuration file*

```
POACxx
{
        Switch = "poa";
        ShlibName = "it_poa_cxx_ibe";
        ShlibMajorVersion = "1";
        IsDefault = "NO";
        PresetOptions = "-t";

#       Header and StubExtension set the generated files
   extension
#       The Default is .cxx and .hh
#
#       StubExtension = "cxx";
#       HeaderExtension = "hh";
};

IFR
{
        Switch = "R";
        ShlibName = "it_ifr_ibe";
        ShlibMajorVersion = "1";
        IsDefault = "NO";
        PresetOptions = "";
};

PSSDLCxx
{
        Switch = "psdl";
        ShlibName = "it_pss_cxx_ibe";
        ShlibMajorVersion = "1";
        IsDefault = "NO";
        PresetOptions = "-t";
        UsePSSDLGrammar = "YES";

#       Header and StubExtension set the generated files
   extension
#       The Default is .cxx and .hh
#
#       StubExtension = "cxx";
#       HeaderExtension = "hh";
};
```

Figure 52: *Distributed IDL configuration file*

```
PSSRCxx
{
        Switch = "pss_r";
        ShlibName = "it_pss_r_cxx_ibe";
        ShlibMajorVersion = "1";
        IsDefault = "NO";
        PresetOptions = "-t";
        UsePSSDLGrammar = "YES";

#       Header and StubExtension set the generated files
   extension
#       The Default is .cxx and .hh
#
#       StubExtension = "cxx";
#       HeaderExtension = "hh";
};

# Java Config Information
Java
{
    Switch = "jbase";
    ShlibName = "idl_java";
    ShlibMajorVersion = "1";
    IsDefault = "NO";
};

POAJava
{
    Switch = "jpoa";
    ShlibName = "jpoa";
    ShlibMajorVersion = "1";
    IsDefault = "NO";

};
```

Given this configuration, you can issue the following idl commands on the IDL file `foo.idl`:

`idl -base foo.idl`	Generates client stub and header code.
`idl -poa foo.idl`	Generates POA code.
`idl -base -poa foo.idl`	Generates code for both client stub and header code and POA code.

APPENDIX B

IONA Policies

Orbix supports a number of proprietary policies in addition to the OMG policies. To create a policy of the proper type you must know the policy's tag.

In this appendix

This appendix contains the following sections:

Client Side Policies

BindingEstablishmentPolicy

Policy Tag

`IT_CORBA::BINDING_ESTABLISHMENT_POLICY_ID`

Data Values

A client's `BindingEstablishmentPolicy` is determined by the members of its `BindingEstablishmentPolicyValue`, which is defined as follows:

```
struct BindingEstablishmentPolicyValue
{
    TimeBase::TimeT relative_expiry;
    unsigned short  max_binding_iterations;
    unsigned short  max_forwards;
    TimeBase::TimeT initial_iteration_delay;
    float           backoff_ratio;
};
```

See Also

"BindingEstablishmentPolicy" on page 262

RelativeBindingExclusiveRoundtripTimeoutPolicy

Policy Tag

`IT_CORBA::RELATIVE_BINDING_EXCLUSIVE_ROUNDTRIP_TIMEOUT_POLICY_ID`

Data Values

This policy's value is set in 100-nanosecond units.

See Also

"RelativeBindingExclusiveRoundtripTimeoutPolicy" on page 265

RelativeBindingExclusiveRequestTimeoutPolicy

Policy Tag

`IT_CORBA::RELATIVE_BINDING_EXCLUSIVE_REQUEST_TIMEOUT_POLICY_ID`

Data Values

This policy's value is set in 100-nanosecond units.

See Also

"RelativeBindingExclusiveRequestTimeoutPolicy" on page 265

RelativeConnectionCreationTimeoutPolicy

Policy Tag

`IT_CORBA::RELATIVE_CONNECTION_CREATION_TIMEOUT_POLICY_ID`

Data Values

The policy's value is set in 100-nanosecond units.

See Also

"RelativeConnectionCreationTimeoutPolicy" on page 265

InvocationRetryPolicy

Policy Tag

`IT_CORBA::INVOCATION_RETRY_POLICY_ID`

Data Values

A client's `InvocationRetryPolicy` is determined by the members of its `InvocationRetryPolicyValue`, which is defined as follows:

```
struct InvocationRetryPolicyValue
{
    unsigned short  max_retries;
    unsigned short  max_rebinds;
    unsigned short  max_forwards;
    TimeBase::TimeT initial_retry_delay;
    float           backoff_ratio;
};
```

See Also

"InvocationRetryPolicy" on page 265

POA Policies

ObjectDeactivationPolicy

Policy Tag

`IT_PortableServer::OBJECT_DEACTIVATION_POLICY_ID`

Data Values

Three settings are valid for this policy:

`DELIVER`**`(default)`**	The object deactivates only after processing all pending requests, including any requests that arrive while the object is deactivating.
`DISCARD`	The POA rejects incoming requests with an exception of `TRANSIENT`. Clients should be able to reissue discarded requests.
`HOLD`	Requests block until the object deactivates. A POA with a `HOLD` policy maintains all requests until the object reactivates. However, this policy can cause deadlock if the object calls back into itself.

See Also

"Setting deactivation policies" on page 279

PersistentModePolicy

Policy Tag

`IT_PortableServer::PERSISTENCE_MODE_POLICY_ID`

Data Values

The only valid value for this policy is `IT_PortableServer::DIRECT_PERSISTENCE`.

See Also

"Direct persistence" on page 206

WellKnownAddressingPolicy

Policy Tag

`IT_CORBA::WELL_KNOWN_ADDRESSING_POLICY_ID`

Data Values

This policy takes a string that maps to the prefix of the configuration variable listing the well known address.

See Also

"Direct persistence" on page 206

WorkQueuePolicy

Policy Tag

`IT_WorkQueue::WORK_QUEUE_POLICY_ID`

Data Values

This policy takes a `WorkQueue` object.

See Also

"Creating the WorkQueue" on page 227

Security Policies

For more detailed information on the following policies see the *CORBA SSL/TLS Guide*.

SessionCachingPolicy

Policy Tag

`IT_TLS_API::TLS_SESSION_CACHING_POLICY`

Data Values

The following settings are valid for this policy:

`CACHE_NONE(default)`	The ORB does not cache session data.
`CACHE_CLIENT`	The ORB will cache session data for client side of a connection.
`CACHE_SERVER`	The ORB will cache session data for server side of a connection.
`CACHE_SERVER_AND_CLIENT`	The ORB stores session information for both the client and server side of a connection.

MaxChainLengthPolicy

Policy Tag

`IT_TLS_API::TLS_MAX_CHAIN_LENGTH_POLICY`

Data Values

This policy takes an integer.

CertContraintsPolicy

Policy Tag

`IT_TLS_API::TLS_CERT_CONSTRAINTS_POLICY`

Data Values

This policy takes an `IT_TLS_API::CertConstraints` object.

CertValidatorPolicy

Policy Tag

`IT_TLS_API::TLS_CERT_VALIDATOR_POLICY`

Data Values

This policy takes a `IT_TLS::CertValidator` object.

Firewall Proxy Policies

For more information on the firewall proxy service see the *Application Server Platform Administrator's Guide*.

InterdictionPolicy

Policy Tag

`IT_FPS::INTERDICTION_POLICY_ID`

Data Values

`PROCEED` **`(default)`** This is the default behavior of the firewall proxy service plug-in. A POA with its `INTERDICTION` policy set to `PROCEED` will be proxified.

`PREVENT` This setting tells the firewall proxy service plug-in to not proxify the POA. POAs with their `INTERDICTION` policy set to `PREVENT` will not use the firewall proxy service and requests made on objects under its control will come directly from the requesting clients.

Index

D

E

F

G

H

P

Q

R

S

T

U

V

W