IONA Orbix 2000
Programmer's Guide
C++ Edition

IONA Technologies PLC
August 2001

M 2 6 7 1

Contents

Preface

Orbix 2000 is a full implementation from IONA Technologies of the Common Object Request Broker Architecture (CORBA), as specified by the Object Management Group. Orbix 2000 complies with the following specifications:

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

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.

Orbix 2000 documentation is periodically updated. New versions between releases are available at this site:

`http://www.iona.com/docs/orbix2000.html`

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 `doc-feedback@iona.com`.

Audience

The *Orbix 2000 Programmer's Guide* is intended to help you become familiar with Orbix 2000, and show how to develop distributed applications using Orbix components. This guide assumes that you are familiar with programming in C++.

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.

Document Conventions

This guide uses the following typographical conventions:

`Fixed-width`	Fixed-width 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 *new terms*.
`italic`	Italicized fixed-width font in syntax and in text denotes variables that you supply, such as arguments to commands, or path names. For example: `% cd /users/`*`your-name`*

This guide may use the following keying conventions:

%	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
>	The notation > represents the DOS, Windows NT, Windows95, or Windows98 command prompt.
...	Ellipses in sample code and syntax descriptions indicate that material has been eliminated to simplify a discussion.
[ ]	Italicized 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.

1 Introduction to Orbix 2000

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.

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.

Orbix is 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.

Why 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 OrbixCOMet, 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.

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.

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.

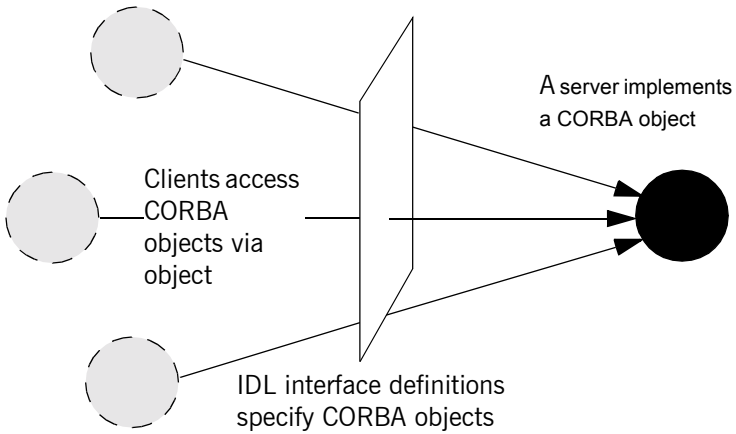

Figure 1: *The nature of abstract CORBA objects*

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.

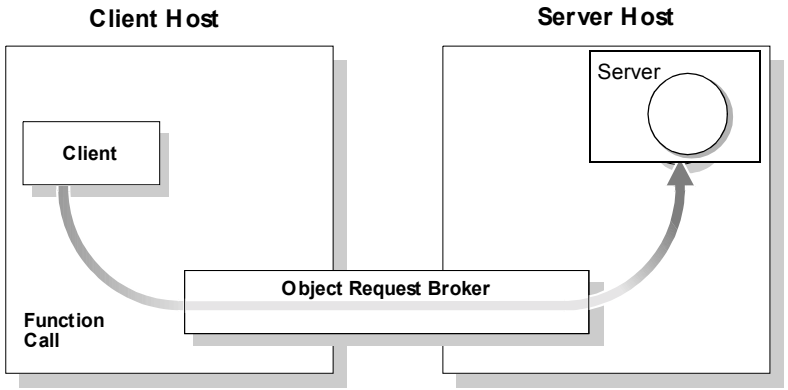

Figure 2: *The object request broker*

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.

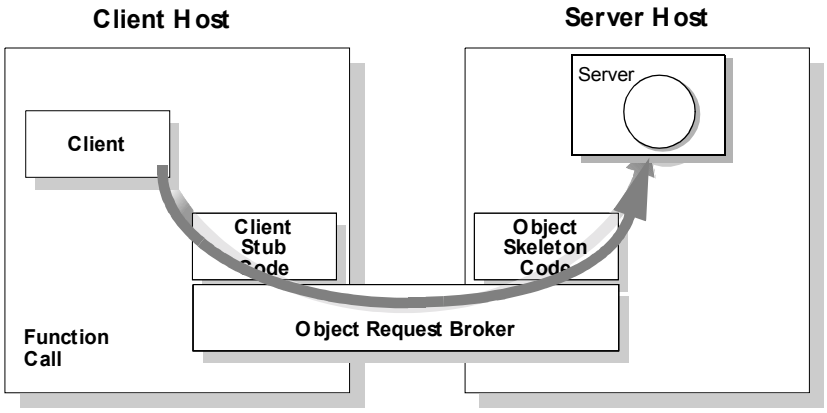

Figure 3: *Invoking on a CORBA object*

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 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.

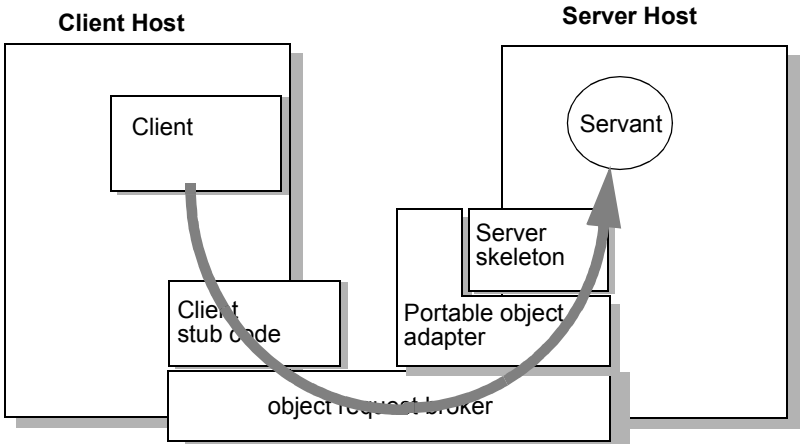

Figure 4: *The portable object adapter*

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 Orbix 2000 developer's kit 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 *Orbix 2000 Code Generation 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.

Of course, multi-threaded applications must themselves be thread-safe. This usually means they need to use thread-synchronization objects such as mutexes or semaphores. Although most platforms provide similar thread synchronization facilities, the interfaces vary widely. Orbix includes an object-oriented thread synchronization portability library which allows you to write portable multi-threaded code.

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.

Asynchronous Messaging and Quality of Service

Orbix implements two key parts of the CORBA messaging specification that are included in CORBA 3.0.

- Asynchronous messaging interfaces allow easy, type-safe asynchronous calls to normal CORBA operations. This means that clients can make a request on a remote service, and then carry on with other work until the reply is ready.
- ORB quality-of-service policies provide finer standardized control over 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 2000 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 2000 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 2000 SSL/TLS components and applications.

COMet

OrbixCOMet 2000 provides a high performance dynamic bridge that enables transparent communication between COM/Automation clients and CORBA servers.

OrbixCOMet 2000 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.

Persistent State Service

Orbix includes the first implementation of the persistent state service (PSS). PSS interposes a CORBA-based abstraction layer between a server and its persistent storage. Orbix's implementation of PSS is based on Berkeley DB, an efficient embedded database that is included with Orbix. By adding new PSS driver plug-ins, applications that use PSS can be reconfigured to store their data in any database without code changes or recompilation.

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.

2 Getting Started with Orbix 2000

You can use the Orbix 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 makefiles. You then complete the distributed application by filling in the missing business logic.

Prerequisites

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

- The Orbix developer's kit is installed on your host.
- Orbix is configured to run on your host platform.

The *Orbix 2000 Administrator's Guide* contains more information on Orbix configuration, and details of Orbix command line utilities.

Setting the Orbix Environment

The scripts that set the Orbix environment are associated with a particular *domain*, which is the basic unit of Orbix configuration. Consult the *Orbix 2000 Installation Guide*, and the *Orbix 2000 Administrator's Guide* for further details on configuring Orbix.

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

Windows

```
> OrbixInstallDir\bin\DomainName_env.bat
```

UNIX

```
% . OrbixInstallDir/bin/DomainName_env.sh
```

OrbixInstallDir is the root directory where Orbix is installed and *DomainName* is the name of an Orbix configuration domain (usually `Orbix2000`).

Hello World Example

This chapter shows how to create, build, and run a complete client/server demonstration with the help of the Orbix Code Generation Toolkit. The architecture of this example system is shown in Figure 5.

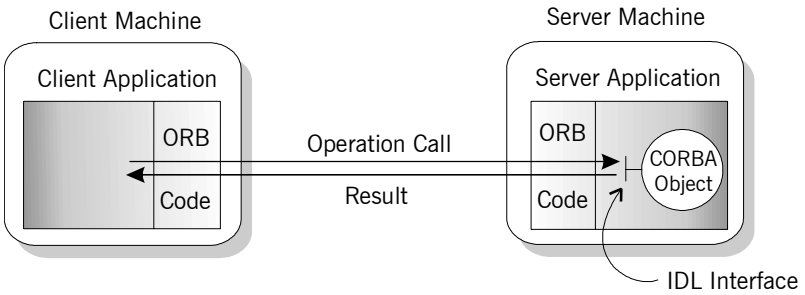

Figure 5: *Client makes a single operation call on a server*

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 C++ 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 Using the Client/Server Wizard

On the Windows NT platform, Orbix provides a wizard add-on to the Microsoft Visual Studio integrated development environment (IDE) that enables you to generate starting point code for Orbix applications.

If you are not working on a Windows platform or if you prefer to use a command line approach to development, see "Development from the Command Line" on page 28.

Ordinarily, the client/server wizard is installed at the same time as Orbix. If the wizard is not on your system, however, consult the Orbix *Install Guide* for instructions on how to install it.

Steps to Implement the Hello World! Application

Implement the Hello World! application with the following steps:

1. Define the IDL interface, `Hello`.
2. Generate the server.
3. Complete and build the server program.

   Implement the single IDL `getGreeting()` operation.
4. Generate the client.
5. Complete and build the client program.

   Insert a line of code to invoke the `getGreeting()` operation.

6. Run the demonstration.

Step 1—Define the IDL Interface

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

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

Create an IDL file `C:\OCGT\HelloExample\hello.idl` using a text editor.

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

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

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

Step 2—Generate the Server

Generate files for the server application using the Orbix Code Generation Toolkit.

To create a server project using the IONA Orbix 2000 client/server wizard:

1. Open the Microsoft Visual C++ 6.0 integrated development environment (IDE).
2. Select **File**→**New** from the Visual C++ menus. A **New** dialog box appears, as shown in Figure 6. Click on the **Projects** tab.
3. Under the **New**→**Projects** tab, select the **IONA Orbix 2000 Client/Server Wizard**. In the **Project name** text box enter `server`, and under the **Location** text box enter `C:\OCGT\HelloExample\server`.

4. When you have finished filling in the text boxes, click **OK** to continue.

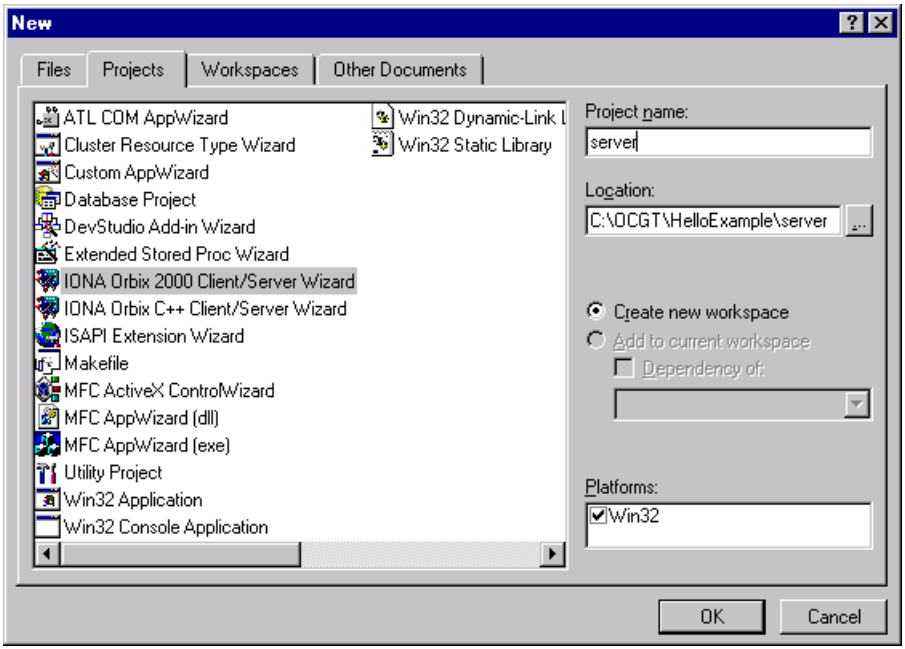

Figure 6: *Starting up the IONA Orbix 2000 client/server wizard*

5. The client/server wizard appears on your screen, as shown in Figure 7. In answer to the question **What CORBA IDL file would you like to use for this project?** enter the location of `hello.idl` in the text box or use the **Browse** button.
6. In answer to the question **Would you like to generate a working client or server?** click on the **Server** radiobutton.
7. Click **Next** to advance to the next screen.
8. The second screen of the server wizard is shown in Figure 8. You can accept the default settings on this screen. Click **Finish** to proceed with generating the server project

   .

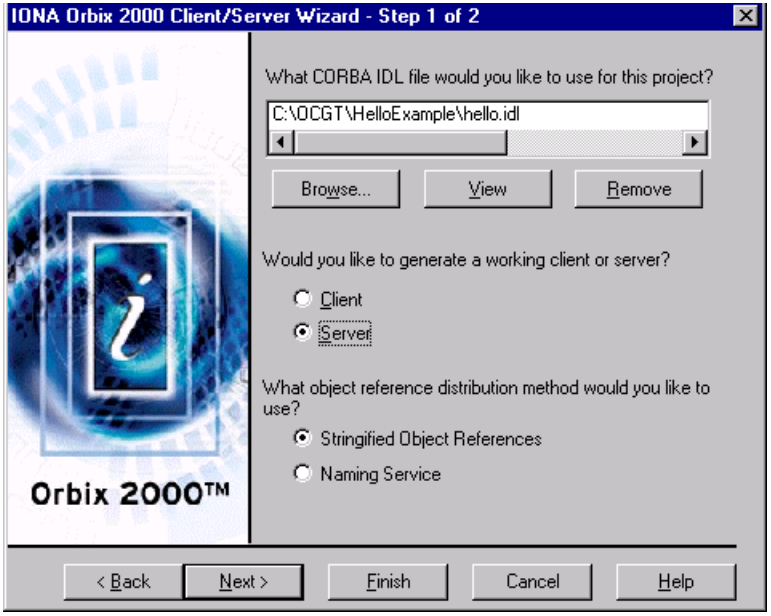

Figure 7: *Step 1 of the wizard for generating an Orbix server*

9. A scrollbox entitled **New Project Information** appears that informs you about the files that were generated. Select **OK** to continue after you have browsed the information.
10. The server workspace has now been generated. Figure 9 shows a list of the source files that have been generated for the server project.
11. Double-click on the `ReadmeOrbix2000Server.txt` file and read the notes contained in it.

Step 3—Complete and Build the Server Program

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

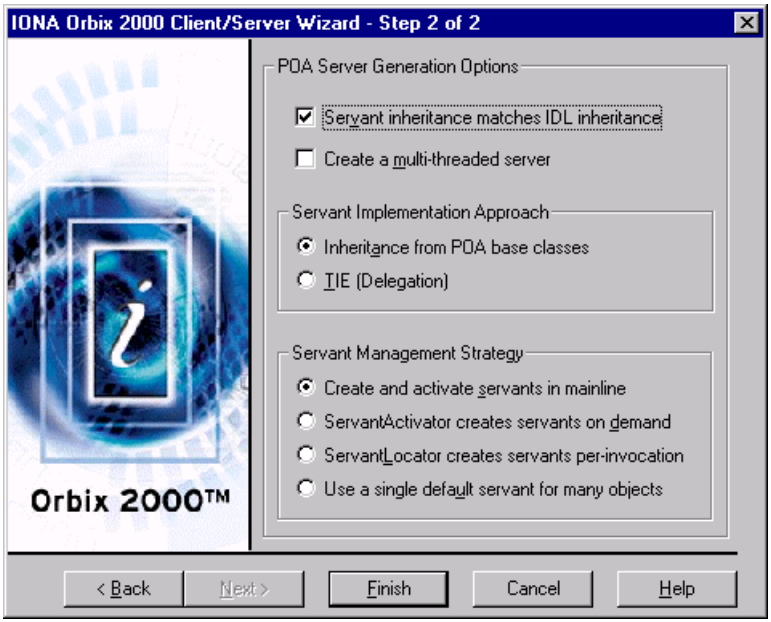

Figure 8: *Step 2 of the wizard for generating an Orbix server*

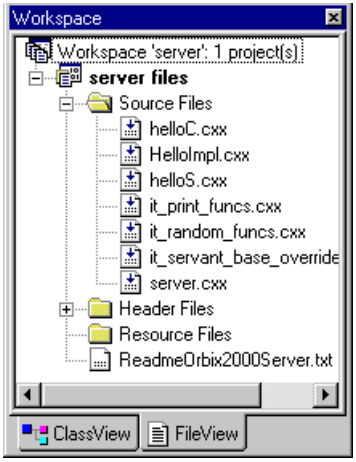

Figure 9: *Workspace for the generated server project*

Edit HelloImpl.cxx

Delete the generated boilerplate code that occupies the body of the `HelloImpl::getGreeting()` function and replace it with the line of code highlighted in bold font below:

```
//C++
...
char*
HelloImpl::getGreeting()
{
    char*                           _result;

    _result = CORBA::string_dup("Hello World!");

    return _result;
}
...
```

The function `CORBA::string_dup()` allocates a copy of the string on the free store. This is needed to be consistent with the style of memory management used in CORBA programming.

Build the Server

From within the Visual C++ IDE select **Build→Build server.exe** to compile and link the server.

By default, the project builds with debug settings and the server executable is stored in `C:\OCGT\HelloExample\server\Debug\server.exe`.

Close the server workspace by selecting **File→Close Workspace**.

Step 4—Generate the Client

Generate files for the client application using the Orbix Code Generation Toolkit.

Windows

To create a client project using the IONA Orbix 2000 client/server wizard:

1. Open the Microsoft Visual C++ 6.0 IDE.

2. Select **File→New** from the Visual C++ menus. A **New** dialog box appears, as shown in Figure 10.
3. Under the **New→Projects** tab, select the **IONA Orbix 2000 Client/Server Wizard**. In the **Project name** text box enter `client`, and under the **Location** text box enter `C:\OCGT\HelloExample\client`.
4. When you have finished filling in the text boxes, click **OK** to continue.

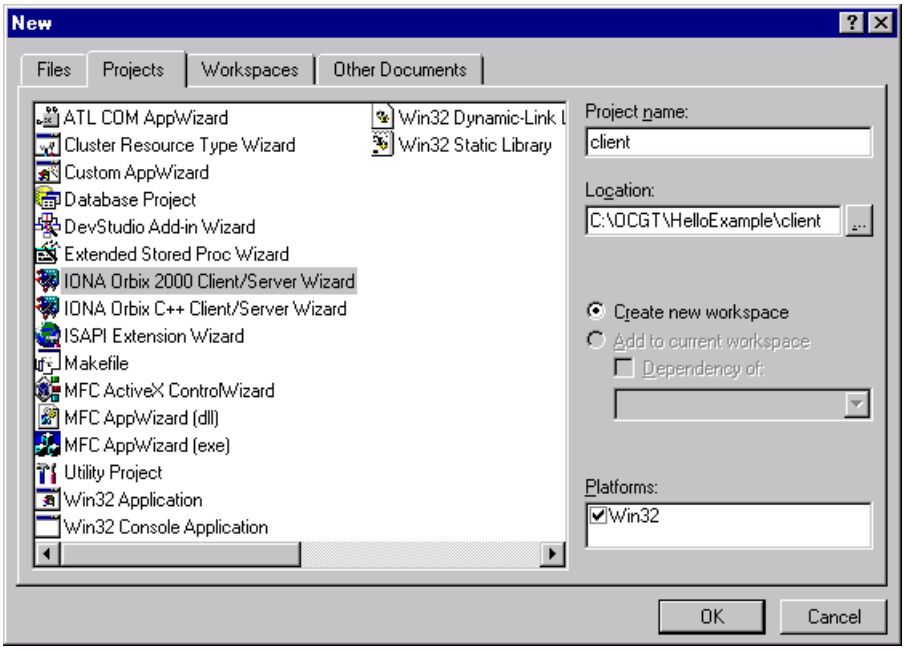

Figure 10: *Starting up the IONA Orbix 2000 client/server wizard*

5. The client/server wizard appears on your screen, as shown in Figure 11. In answer to the question **What CORBA IDL file would you like to use for this project?** enter the location of `hello.idl` in the text box or use the **Browse** button.
6. In answer to the question **Would you like to generate a working client or server?** click on the **Client** radiobutton.
7. Click **Finish** to proceed with generating the client project.

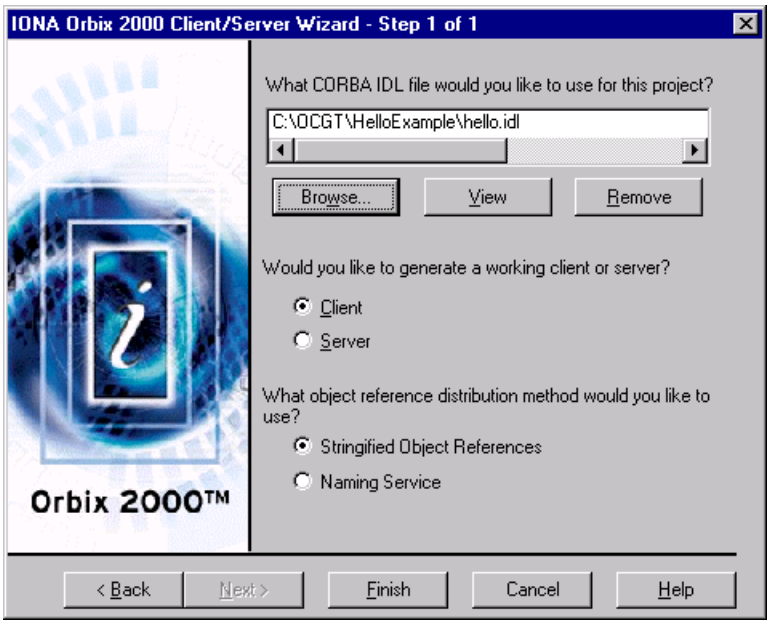

Figure 11: *Step 1 of the wizard for generating an Orbix client*

8. A scrollbox entitled **New Project Information** appears, informing you about the files that were generated. Click **OK** to continue after you have browsed the information.
9. The client workspace has now been generated. Figure 12 shows a list of the source files that have been generated for the client project.
10. Double-click on the file `ReadmeOrbix2000Client.txt` and read the contents of the file.

Step 5—Complete and Build the Client Program

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

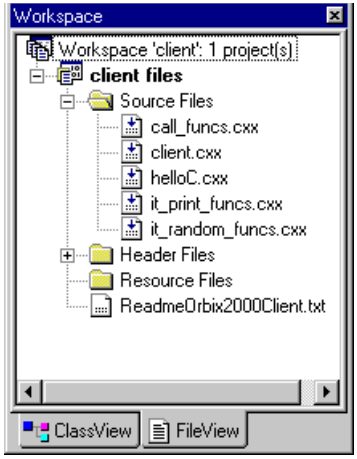

Figure 12: *Workspace for the generated client project*

Edit client.cxx

Search for the line where the `call_Hello_getGreeting()` function is called. Delete this line and replace it with the two lines of code highlighted in bold font below:

```
//C++
//File: 'client.cxx'
...
     if (CORBA::is_nil(Hello1))
     {
          cerr << "Could not narrow reference to interface "
               << "Hello" << endl;
     }
     else
     {
          CORBA::String_var strV = Hello1->getGreeting();
          cout << "Greeting is:  " << strV << endl;
     }
...
```

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()` function in the server application.

The returned string is put into a C++ object, `strV`, of the type `CORBA::String_var`. The destructor of this object will delete the returned string so that there is no memory leak in the above code.

Build the Client

From within the Visual C++ IDE select **Build→Build client.exe** to compile and link the client.

By default, the project will build with debug settings and the client executable will be stored in `C:\OCGT\HelloExample\client\Debug\client.exe`.

Close the client workspace by selecting **File→Close Workspace**.

Step 6—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 new MS-DOS prompt.

   > start_*DomainName*_services.bat

   Where *DomainName* is the name of the default configuration domain (usually `orbix2000`).

2. Run the server program.

   Open a new MS-DOS prompt.

   ```
   > cd C:\OCGT\HelloExample\server\Debug
   > server.exe
   ```

   The server outputs the following lines to the screen:

   ```
   Initializing the ORB
   ```

```
Writing stringified object reference to Hello.ref
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 file `C:\temp\Hello.ref`.
- The server opens an IP port and begins listening on the port for connection attempts by CORBA clients.

3. Run the client program.

   Open a new MS-DOS prompt.

   ```
   > cd C:\OCGT\HelloExample\client\Debug
   > client.exe
   ```

   The client outputs the following lines to the screen:

   ```
   Client using random seed 0
   Reading stringified object reference from Hello.ref
   Greeting is: Hello World!
   ```

   The client performs the following steps when it is run:

   - It reads the stringified object reference for the `Hello` object from the `C:\temp\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.

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`_*DomainName*_**`services`**

   Where *DomainName* is the name of the default configuration domain (usually `orbix2000`).

Development from the Command Line

Starting point code for Orbix client and server applications can also be generated using the `idlgen` command line utility, which offers equivalent functionality to the client/server wizard presented in the previous section.

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

Steps to Implement the Hello World! Application

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.

   Implement the single IDL `getGreeting()` operation.
4. Complete the client program.

   Insert a line of code to invoke the `getGreeting()` operation.
5. Build and run the demonstration.

Step 1—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.

Step 2—Generate Starting Point Code.

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

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

```
idlgen cpp_poa_genie.tcl -all hello.idl
```

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

```
hello.idl:
cpp_poa_genie.tcl: creating it_servant_base_overrides.h
cpp_poa_genie.tcl: creating it_servant_base_overrides.cxx
cpp_poa_genie.tcl: creating HelloImpl.h
cpp_poa_genie.tcl: creating HelloImpl.cxx
cpp_poa_genie.tcl: creating server.cxx
cpp_poa_genie.tcl: creating client.cxx
cpp_poa_genie.tcl: creating call_funcs.h
cpp_poa_genie.tcl: creating call_funcs.cxx
cpp_poa_genie.tcl: creating it_print_funcs.h
cpp_poa_genie.tcl: creating it_print_funcs.cxx
cpp_poa_genie.tcl: creating it_random_funcs.h
cpp_poa_genie.tcl: creating it_random_funcs.cxx
cpp_poa_genie.tcl: creating Makefile
```

The files you can edit to customize the client and server applications are:

Table 1: *Main C++ source files for the Hello World! application*

Client Files	Server Files

`client.cxx`	`server.cxx`
	`HelloImpl.h`
	`HelloImpl.cxx`

Step 3—Complete the Server Program

Complete the implementation class, `HelloImpl`, by providing the definition of the `HelloImpl::getGreeting() member function` . This C++ function provides the concrete realization of the `Hello::getGreeting()` IDL operation.

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

```
//C++
//File 'HelloImpl.cxx'
...
char *
HelloImpl::getGreeting() throw(
    CORBA::SystemException
)
{
    char *                          _result;

    _result = CORBA::string_dup("Hello World!");

    return _result;
}
...
```

The function `CORBA::string_dup()` allocates a copy of the `"Hello World!"` string on the free store. It would be an error to return a string literal directly from the CORBA operation because the ORB automatically deletes the return value after the function has completed. It would also be an error to create a copy of the string using the C++ new operator.

Step 4—Complete the Client Program

Complete the implementation of the client `main()` function in the `client.cxx` 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.cxx` file and search for the line where the `call_Hello_getGreeting()` function is called. Delete this line and replace it with the two lines of code highlighted in bold font below:

```
//C++
//File: 'client.cxx'
...
     if (CORBA::is_nil(Hello1))
     {
          cerr << "Could not narrow reference to interface "
               << "Hello" << endl;
     }
     else
     {
          CORBA::String_var strV = Hello1->getGreeting();
          cout << "Greeting is:  " << strV << endl;
     }
...
```

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()` function in the server application.

The returned string is put into a C++ object, `strV`, of the type `CORBA::String_var`. The destructor of this object will delete the returned string so that there is no memory leak in the above code.

Step 5—Build and Run the Demonstration

The `Makefile` generated by the code generation toolkit has a complete set of rules for building both the client and server applications.

To build the client and server:

Windows

At a command-line prompt, from the `C:\OCGT\HelloExample` directory enter:

```
> nmake
```

UNIX

At a command-line prompt, from the `OCGT/HelloExample` directory enter:

```
% make -e
```

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

   `start`_*DomainName*_**`services`**

   Where *DomainName* is the name of the default configuration domain (usually `orbix2000`).

2. Run the server program.

   Open a new MS-DOS prompt, or `xterm` window (UNIX). From the `C:\OCGT\HelloExample` directory enter the name of the executable file—`server.exe` (Windows) or `server` (UNIX).The server outputs the following lines to the screen:

   ```
   Initializing the ORB
   Writing stringified object reference to Hello.ref
   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.

3. Run the client program.

   Open a new MS-DOS prompt, or `xterm` window (UNIX). From the `C:\OCGT\HelloExample` directory enter the name of the executable file—`client.exe` (Windows) or `client` (UNIX).

   The client outputs the following lines to the screen:

```
Client using random seed 0
Reading stringified object reference from Hello.ref
Greeting is: Hello World!
```

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.

4. When you are finished, terminate all processes.

   Shut down the server 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`_*DomainName*_**`services`**

   Where *DomainName* is the name of the default configuration domain (usually `orbix2000`).

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 18).

3 First Application

This chapter uses sample code to show how to develop an enterprise application with Orbix.

Orbix enterprise applications consist of CORBA objects with clearly defined interfaces that can be accessed across a network.

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.

The application described here performs these tasks:

- A server program creates a single object that represents a building such as a warehouse.
- A client program uses the object's interface to get the building's address, check its availability, and reserve it for specific dates.

To recreate this program, you must have installed and configured Orbix for your particular platform.

This chapter covers the following topics:

- Overview of the development process.
- Development steps.
- Learning more about the server.

Overview of the Development Process

The Orbix Code Generation Toolkit can ease the process of application development for Orbix programmers, but its use is not compulsory. This section outlines the responsibilities of client developers and server developers in cases where the code generation toolkit is *not* used and in cases where it *is* used.

Development Without Using Code Generation

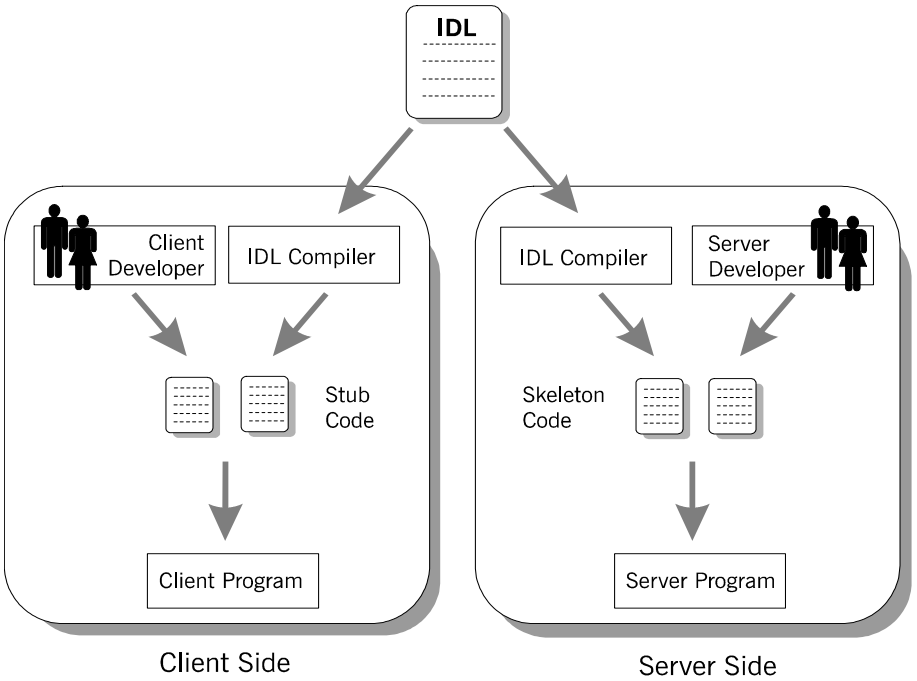

Figure 13: *Development overview without using code generation*

The first step in the development process is to 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.

The development process on the client side is illustrated on the left of Figure 13. The steps are:

- 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.
- The client developer writes the rest of the client application from scratch.

- The client developer builds the application.

  Typically, the developer has to write a customized makefile to build the client program.

The development process on the server side is illustrated on the right of Figure 13. The steps are:

- 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.
- The server developer writes the rest of the server application from scratch.

  An implementation class must be written by the server developer for each interface appearing in the IDL file.
- The server developer builds the application.

  Typically, the developer has to write a customized makefile to build the server program.

Development Using Code Generation

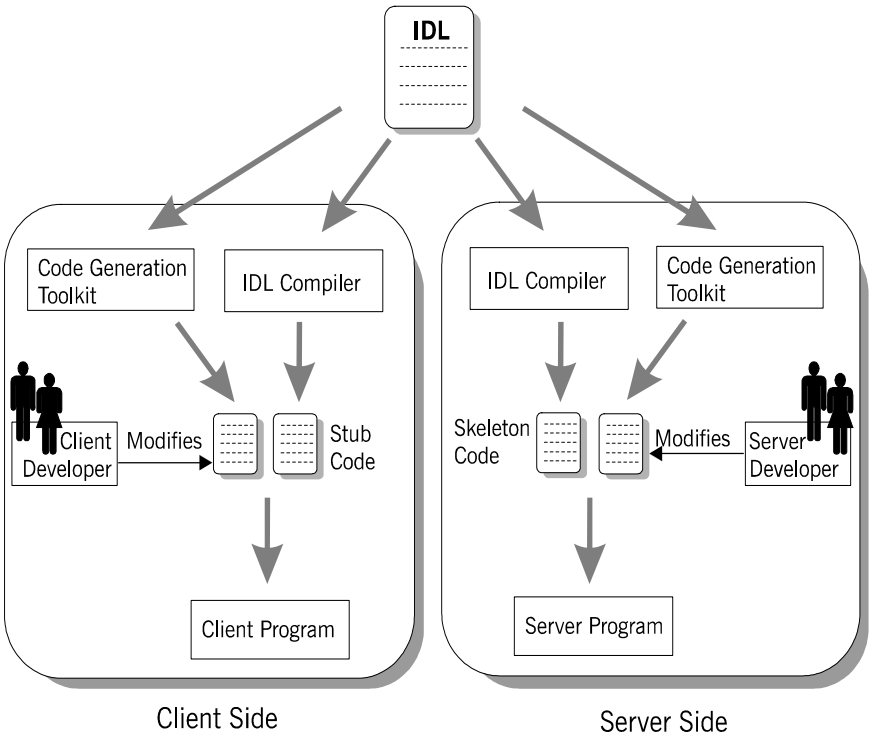

Figure 14: *Development overview using code generation*

Using the code generation toolkit, a large proportion of the code required for the client and server programs is generated automatically. The toolkit takes the IDL file as input and, based on the declarations in the IDL file, generates a complete, working Orbix application.

Developers can then modify the generated code to add business logic to the application.

The development process on the client side, using code generation, is illustrated on the left of Figure 14. The steps are:

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. The client developer can modify the dummy client to complete the application.

   The client developer does not have to write boilerplate CORBA code.
4. The client developer builds the application.

   A makefile is generated by the code generation toolkit.

The development process on the server side, using code generation, is illustrated on the right of Figure 14. The steps are:

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. The server developer can modify the dummy server to complete the application logic.

   The server developer does not have to write boilerplate CORBA code.

   The implementations of IDL interfaces can be modified by adding business logic to the class definitions.
4. The server developer builds the application.

   A makefile is generated by the code generation toolkit.

Locating CORBA Objects

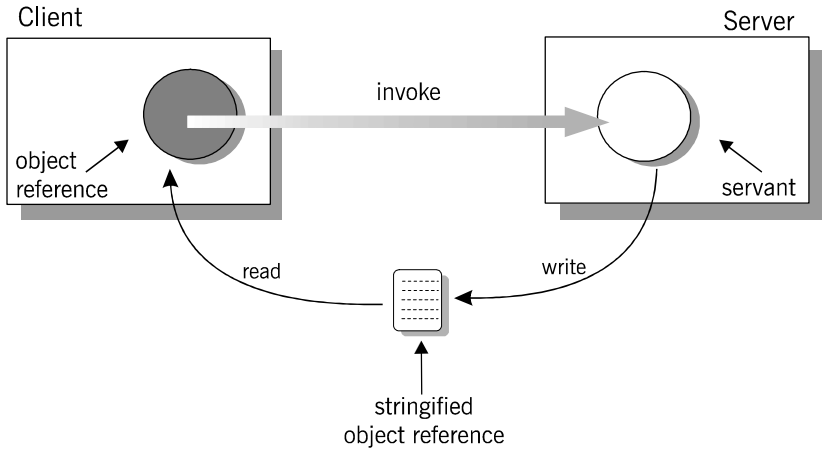

Figure 15: *Simple strategy for passing object references to clients*

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 15, 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.

This approach is convenient for simple demonstrations but is not recommended for use in realistic applications. The CORBA naming service, described in Chapter 18 on page 377, provides a more sophisticated and scalable approach to distributing object references.

Development Steps

To develop an Orbix application:

1. Define the 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 C++ header and 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. The server developer 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 and run the application.

Step 1—Define the IDL Interfaces

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 following IDL code 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:

```
//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 code can be explained as follows:

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.

Step 2—Generate Starting Point Code

The recommended way to begin a CORBA application is to use the code generation toolkit to generate starting point code. The toolkit contains two key components:

- The `idlgen` interpreter.

  This is an executable file that processes IDL files based on the instructions contained in predefined code generation scripts.
- A set of code generation scripts, or *genies*.

  A number of genies are supplied with the toolkit. Most important of these is the `cpp_poa_genie.tcl` genie that is used to generate starting point code for a C++ application.

Taking the `building.idl` IDL file as input, the `cpp_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 cpp_poa_genie.tcl -all 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 main client file generated by the `cpp_poa_genie.tcl` genie is:

`client.cxx`	Implementation of the client.

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

`server.cxx`	Server `main()` containing the server initialization code.
`BuildingImpl.h`	Header file that declares the `BuildingImpl` servant class.

`BuildingImpl.cxx`	Implementation of the `BuildingImpl` servant class.
`it_servant_base_overrides.h`	Header file that declares a base class for all servant classes. See page 239.
`it_servant_base_overrides.cxx`	Implementation of the base class for all servant classes. See page 239.

A makefile is generated for building the application:

`Makefile`	The generated makefile defines rules to build both the client and the server.

The following files are also generated and support a dummy implementation of the client and server programs:

```
call_funcs.h
call_funcs.cxx
it_print_funcs.h
it_print_funcs.cxx
it_random_funcs.h
it_random_funcs.cxx
```

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 that prints the `in` and `inout` parameters to the screen when it is invoked. 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.

Step 3—Compile the IDL Definitions

Note: This step is optional when developing an application using the code generation toolkit. The `Makefile` generated by the toolkit has a rule to run the IDL compiler automatically.

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

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

```
idl -base -poa building.idl
```

The `-base` option generates client stub and header code in C++. The `-poa` option generates server-side code for the portable object adapter (POA).

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 C++ files when it compiles the `building.idl` file. These files contain C++ 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.

  This code is compiled and linked with client programs to enable them to make remote invocations on `Building` CORBA objects.
- Server skeleton code.

  This code is compiled and linked with server programs to enable them to service invocations on `Building` CORBA objects.

Client Stub Code

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

`building.hh`	A header file containing definitions that correspond to the various IDL type definitions. Client source code must include this file using a `#include` preprocessor directive.
`buildingC.cxx`	A file containing code that enables remote access to `Building` objects. This file must be compiled and linked with the client executable.

Any clients that want to invoke on CORBA objects that support the `Building` interface must include the header file `building.hh` and link with the stub code `buildingC.cxx`.

Server Skeleton Code

The skeleton code is used by servers and consists of the following files:

`buildingS.hh`	A header file containing type definitions for implementing servant classes. Server source code must include this file using a `#include` preprocessor directive.
`buildingS.cxx`	A file containing skeleton code that enables servers to accept calls to `Building` objects. This file must be compiled and linked with the server executable.
`building.hh`	A header file common to client stub code and server skeleton code. This file is included by `buildingS.hh`, so server files do not need to explicitly include it.

`buildingC.cxx`	Source file common to client stub code and server skeleton code. This file must be compiled and linked with the server executable.

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.

Server files include the `buildingS.hh` header file, which recursively includes the file `building.hh`. The server must be linked with both `buildingC.cxx` and `buildingS.cxx`.

IDL to C++ Mapping

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

Both the client and the server source files include the generated header file `building.hh`, which contains the C++ mappings for the `Building` interface (see "Step 1—Define the IDL Interfaces" on page 42):

```
   //C++
1  class Building : public virtual CORBA::Object
   {
     public:
       ...
2      virtual char* address() = 0;
       ...
3      virtual CORBA::Boolean available(CORBA::Long date) = 0;
4      virtual CORBA::Boolean reserveDate(
           CORBA::Long date,
           CORBA::Long_out  confirmation
       ) = 0;
       ...
   };
```

The code can be explained as follows:

1. The `Building` class defines proxy objects for the `Building` interface. This class includes member methods that correspond to the attributes and operations of the IDL interface. When a client program calls methods on an object of type `Building`, Orbix forwards the method calls to a server object that supports the `Building` interface.
2. The C++ pure virtual member method `address()` maps to the readonly IDL string attribute `address`. Clients call this method to get the attribute's current value, which returns the C++ type `char*`.
3. The pure virtual C++ member method `available()` maps to the IDL operation of the same name. It returns type `CORBA::Boolean`, which maps to the equivalent IDL type `boolean`. Its single parameter is of `CORBA::Long` type, which is a `typedef` of a basic C++ integer type. This maps to the operation parameter of IDL type `long`.
4. The operation `reserveDate()` has one input parameter, `date`, and one output parameter, `confirmation`, both of IDL type `long`. The return type is `CORBA::Boolean`. Input parameters (specified as IDL `in` parameters) are passed by value in C++.

   Output parameters are passed by reference. Every CORBA data type has a corresponding `_out` type that is used to declare output parameters. For basic types, such as `short` and `long`, the `_out` type is a `typedef` of a reference to the corresponding C++ type. For example, the `CORBA::Long_out` type is defined in the CORBA namespace as:

   ```
   typedef CORBA::Long& CORBA::Long_out;
   ```

Other helper data types and methods generated in this file are described when they are used in this chapter.

Step 4—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`.

For each servant class:

- Declare the servant class.

The code generation toolkit generates an outline servant header file for every interface. The `BuildingImpl` servant class is declared in the header file `BuildingImpl.h`.

- Define the servant class.

  The code generation toolkit generates a dummy definition of every servant class. The `BuildingImpl` servant class is defined in the file `BuildingImpl.cxx`.

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 "Learning More About the Server" on page 61.

Declare the BuildingImpl Servant Class

The code generation toolkit generates a header file, `BuildingImpl.h`, that declares the `BuildingImpl` servant class. You can use this starting point code to implement the `Building` interface.

Note: The name of the `BuildingImpl` servant class is not significant but simply conforms to a naming convention that helps distinguish servant code from other application code.

You can modify the generated code in `BuildingImpl.h` to add member variables needed for the implementation. The code shown here provides a simple implementation of `BuildingImpl`.

Manual additions to the generated code are shown in bold font.

```
//C++
// File: 'BuildingImpl.h'
...
1  #include "buildingS.hh"
   #include "it_servant_base_overrides.h"

2  class BuildingImpl :
       public virtual IT_ServantBaseOverrides,
       public virtual POA_Building
   {
```

```
    public:
        BuildingImpl(PortableServer::POA_ptr);
        virtual ~BuildingImpl();

        // _create() -- create a new servant.
        static POA_Building* _create(PortableServer::POA_ptr);

        // IDL operations
        //
3       virtual CORBA::Boolean available(
            CORBA::Long                     date
        ) IT_THROW_DECL((CORBA::SystemException));

        virtual CORBA::Boolean reserveDate(
            CORBA::Long                     date,
            CORBA::Long_out                 confirmation
        ) IT_THROW_DECL((CORBA::SystemException));

        // IDL attributes
        //
4       virtual char* address()
            IT_THROW_DECL((CORBA::SystemException));

    private:
5       //-----------------------
        // Private Member Variables
        //-----------------------
        CORBA::Long        m_confirmation_counter;
        CORBA::Long        m_reservation[366];

        // Instance variables for attributes.
6       CORBA::String_var              m_address;
        ...
    };
```

This code can be described as follows:

1. Servers include the `buildingS.hh` skeleton file, which declares the C++ `POA_Building` class.

   The `POA_Building` class is a class generated by the IDL compiler that allows you to implement the `Building` interface using the inheritance

approach. In general, for any interface, *InterfaceName*, a corresponding class, POA_*InterfaceName*, is generated by the IDL compiler.

2. The BuildingImpl servant class inherits from POA_Building and IT_ServantBaseOverrides.

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

   The IT_ServantBaseOverrides class is used to override the definition of some standard virtual methods. For a discussion of this class, see page 239.

3. A member method declaration is generated for each of the operations in the Building interface.

   The IT_THROW_DECL((*ExceptionList*)) macro is used by Orbix to insulate generated code from variations between C++ compilers. The macro maps to throw(*ExceptionList*) for compilers that support exceptions, or to an empty string, "", for compilers that do not.

4. Member method declarations are generated for each of the attributes in the Building interface.

   Read-only attributes require a single method that returns the current value of the attribute. Read/write attributes require two methods: one that returns the current value, and another that takes an input parameter to set the value.

5. The lines of code shown in bold font are added to the generated code to complete the application. Two additional 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.

6. The m_address is a CORBA string that stores the address of the building.

   The declared type of m_address, CORBA::String_var, is a smart pointer type that functions as a memory management aid. String pointers

declared as `CORBA::String_var` are used in a similar way to plain `char *` pointers, except that it is never necessary to delete the string explicitly.

Note: The code generation toolkit automatically generates a private member `m_address` to represent the state of the IDL `address` attribute. However, this generated class member is *not* part of the standard IDL-to-C++ mapping. It is generated solely for your convenience and you are free to remove this line from the generated code if you so choose.

Define the BuildingImpl Servant Class

The code generation toolkit also generates the `BuildingImpl.cxx` 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 and destructor 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.

```
// C++
// File: 'BuildingImpl.cxx'
...
#include "BuildingImpl.h"
// _create() -- create a new servant.
POA_Building*
1 BuildingImpl::_create(PortableServer::POA_ptr the_poa)
{
    return new BuildingImpl(the_poa);
}

// BuildingImpl constructor
//
// Note: since we use virtual inheritance, we must include an
// initialiser for all the virtual base class constructors that
// require arguments, even those that we inherit indirectly.
//
```

```
BuildingImpl::BuildingImpl(
    PortableServer::POA_ptr the_poa
) :
    IT_ServantBaseOverrides(the_poa),
2   m_address( "200 West Street, Waltham, MA." ),
    m_confirmation_counter(1)
{
    for (int i=0; i<366; i++) { m_reservation[i] = 0; }
}

// ~BuildingImpl destructor.
//
3 BuildingImpl::~BuildingImpl()
{
    // Intentionally empty.
}

// available() -- Implements IDL
//                operation "Building::available()".
//
CORBA::Boolean
BuildingImpl::available(
    CORBA::Long date
) IT_THROW_DECL((CORBA::SystemException))
{
4   if (1<=date && date<=366) {
        return (m_reservation[date-1]==0);
    }

    return 0;
}

// reserveDate() -- Implements IDL
//                  operation "Building::reserveDate()".
//
CORBA::Boolean
BuildingImpl::reserveDate(
    CORBA::Long date,
    CORBA::Long_out confirmation
) IT_THROW_DECL((CORBA::SystemException))
{
5   confirmation = 0;
```

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

    // address() -- Accessor for IDL attribute "Building::address".
    //
    char *
    BuildingImpl::address() IT_THROW_DECL((CORBA::SystemException))
    {
6       return CORBA::string_dup(m_address);
    }
```

The code can be explained as follows:

1. `_create()` is a static member method of `BuildingImpl` that creates `BuildingImpl` instances.

   Note that `_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.

2. 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.

3. The `BuildingImpl` destructor is an appropriate place to free any member variables that were allocated on the heap. In this example it is empty.

4. A few lines of code are added here to implement the `available()` operation. 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).

5. A few lines of code are added here to implement the `reserveDate()` operation. Because `confirmation` is declared as an `out` parameter in IDL, it is passed by reference in C++. The value assigned to it is therefore readable by the code that called `reserveDate()`.

6. `CORBA::string_dup()` is used to allocate a copy of the string `m_address` on the free store.

   It would be an error to return the private string pointer directly from the operation because the ORB automatically deletes the return value after the operation has completed.

   It would also be an error to allocate the string copy using the C++ `new` operator.

Step 5—Develop the Client Program

The generated code in the `client.cxx` 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.

```
//C++
//File: 'client.cxx'
...
#include "building.hh"
...
// global_orb -- make ORB global so all code can find it.
//
CORBA::ORB_var
1 global_orb = CORBA::ORB::_nil();

// read_reference() -- read an object reference from file.
//
```

```
   static CORBA::Object_ptr
2  read_reference(
       const char*      file
   )
   {
       cout << "Reading stringified object reference from "
            << file << endl;
       ifstream ifs(file);
       CORBA::String_var str;
       ifs >> str;
       if (!ifs) {
           cerr << "Error reading object reference from "
                << file << endl;
           return CORBA::Object::_nil();
       }
       return global_orb->string_to_object(str);
   }
   ...

   // main() -- the main client program.
   int
   main(int argc, char **argv)
   {
       int exit_status = 0;
       try
       {
           // For temporary object references.
           CORBA::Object_var tmp_ref;

           // Initialise the ORB.
           //
3          global_orb = CORBA::ORB_init(argc, argv);

           // Exercise the Building interface.
           //
4          tmp_ref = read_reference("Building.ref");
5          Building_var Building1 = Building::_narrow(tmp_ref);
           if (CORBA::is_nil(Building1))
           {
               cerr << "Could not narrow reference to interface "
                    << "Building" << endl;
           }
```

```
            else
            {
6               run_warehouse_menu(Building1);
            }
        }
        catch(CORBA::Exception &ex)
        {
            cerr << "Unexpected CORBA exception: " << ex << endl;
            exit_status = 1;
        }

        // Ensure that the ORB is properly shutdown and cleaned up.
        //
        try
        {
7           global_orb->shutdown(1);
            global_orb->destroy();
        }
        catch (...)
        {
            // Do nothing.
        }
        return exit_status;
    }
```

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 the value `CORBA::ORB::_nil()`, which represents a blank object reference of type `CORBA::ORB_ptr`.

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 `CORBA::ORB::string_to_object()`. The return type of `read_reference()` is `CORBA::Object_ptr`—the base type for object references.

   If there is an error, `read_reference()` returns `CORBA::Object::_nil()`, which represents a blank object reference of type `CORBA::Object_ptr`.

3. Call `CORBA::ORB_init()` to get an object reference to the initialized ORB.

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

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

   The `tmp_ref` variable is of `CORBA::Object_var` type. This is a smart pointer type that automatically manages the memory it references.

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

   The mapping for every interface defines a static member method `_narrow()` that lets you narrow an object reference from a base type to a derived type. It is similar to a C++ dynamic cast operation, but is used specifically for types related via IDL inheritance.

6. Replace the lines of generated code in the `else` clause 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, `1`, indicates that `shutdown()` blocks until shutdown is complete.

   `CORBA::ORB::destroy()` destroys the ORB object and reclaims all resources associated with it.

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 the `Building` class method using the proxy object. The proxy object redirects the C++ calls across the network to the appropriate servant method.

The following code uses the C++ arrow operator (->) on the `Building_ptr` object `warehouse` to access `Building` class methods.

Additions to the code are shown in bold font.

```
//C++
//File: 'client.cxx'
void
run_warehouse_menu(Building_ptr warehouseP)
{
    CORBA::String_var addressV = warehouseP->address();
    cout << "The warehouse address is:" << endl
         << addressV.in() << endl;

    CORBA::Long date;
    CORBA::Long confirmation;
    char quit = 'n';
    do {
        cout << "Enter day to reserve warehouse (1,2,...): ";
        cin >> date;
        if(warehouseP->available(date)) {
            if (warehouseP->reserveDate(date, confirmation) )
                cout << "Confirmation number: "
                     << confirmation << endl;
            else
                cout << "Reservation attempt failed!" << endl;
        }
        else {
            cout << "That date is unavailable." << endl;
        }
        cout << "Quit? (y,n)";
        cin >> quit;
    }
    while (quit == 'n');
}
```

Step 6—Build and Run the Application

The prerequisites for running this application are:

- The Orbix runtime is installed on the machine where the demonstration is run.

- Orbix has been correctly configured. See the *Orbix 2000 Administrator's Guide* for details.

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

Build the Application

The makefile generated by the code generation toolkit has a complete set of rules for building both the client and server applications.

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

Windows

```
> nmake
```

UNIX

```
% make -e
```

Run the Application

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`_*DomainName*_**`services`**

   Where *DomainName* is the name of the default configuration domain (usually `orbix2000`).

2. Run the server program.

   Open a new DOS prompt in Windows, or `xterm` in UNIX. The executable file is called `server.exe` (Windows) or `server` (UNIX).

   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 `CORBA::ORB::run()`.

3. Run the client program.
4. Open a new DOS prompt in Windows, or `xterm` in UNIX. The executable file is called `client.exe` (Windows) or `client` (UNIX).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`_*DomainName*_**`services`**

   Where *DomainName* is the name of the default configuration domain (usually `orbix2000`).

Learning More About the Server

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

1. Create a termination handler object.
2. Initialize the ORB.
3. Create a POA for transient objects.
4. Create servant objects.
5. Activate CORBA objects.

   The default server code activates one CORBA object for each of the interfaces defined in the IDL file.
6. Export object references.

An object reference is exported for each of the activated CORBA objects.

7. Put the ORB into an active state.

   Put the ORB into a state where it is ready to receive and process invocations on CORBA objects.

8. Shut down the ORB.

   The ORB should be shut down cleanly before exiting and any heap-allocated memory should be deleted.

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.cxx` file piece by piece. For a complete source listing of `server.cxx`, see page 72.

Create a Termination Handler Object

Orbix provides its own `IT_TerminationHandler` class, which handles server shutdown in a portable manner.

On UNIX, the termination handler handles the following signals:

```
SIGINT
SIGTERM
SIGQUIT
```

On Windows, the termination handler is just a wrapper around `SetConsoleCtrlHandler`, which handles delivery of the following control events:

```
CTRL_C_EVENT
CTRL_BREAK_EVENT
CTRL_SHUTDOWN_EVENT
CTRL_LOGOFF_EVENT
CTRL_CLOSE_EVENT
```

The main routine can create a termination handler object on the stack. On POSIX platforms, it is critical to create this object in the main thread before creation of any other thread, especially before calling `ORBinit()`, as follows:

```
int
main(int argc, char** argv)
{
    IT_TerminationHandler
```

```
    termination_handler(termination_handler_callback);
   // ...
}
```

You can create only one termination handler object in a program. The server shutdown mechanism and `termination_handler_callback()` are discussed in detail in “Shut Down the ORB” on page 70.

Initialize the ORB

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

```
//C++
...
// global_orb -- make ORB global so all code can find it.
CORBA::ORB_var
1 global_orb = CORBA::ORB::_nil();
...

int
main(int argc, char **argv)
{
    ...
    cout << "Initializing the ORB" << endl;
2   global_orb = CORBA::ORB_init(argc, argv);
    ...
```

The code can be explained as follows:

1. The type `CORBA::ORB_var` is a smart pointer class that can be used to refer to objects of type `CORBA::ORB`. Syntactically, a `CORBA::ORB_var` is similar to the pointer type `CORBA::ORB*`. The advantage of using a smart pointer is that it automatically deletes the memory pointed at as soon as it goes out of scope. This helps to prevent memory leaks.

   The value `CORBA::ORB::_nil()` is an example of a *nil object reference*. A nil object reference is a blank value that can legally be passed as a CORBA parameter or return value.

2. `CORBA::ORB_init()` is used to create an instance of an ORB. Command-line arguments can be passed to the ORB via `argc` and `argv`. `ORB_init()` searches `argv` for arguments of the general form `-ORB`*suffix*, parses these arguments, and removes them from the argument list.

Create a POA for Transient Objects

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

```
//C++
try {
    // For temporary object references.
    CORBA::Object_var tmp_ref;
    ...
1   tmp_ref = global_orb->resolve_initial_references("RootPOA");
2   PortableServer::POA_var root_poa =
        PortableServer::POA::_narrow(tmp_ref);
    assert(!CORBA::is_nil(root_poa));

3   PortableServer::POAManager_var root_poa_manager
        = root_poa->the_POAManager();
    assert(!CORBA::is_nil(root_poa_manager));

    // Now create our own POA.
4   PortableServer::POA_var 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()` on the ORB with the argument `"RootPOA"`.

   `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 the type `PortableServer::POA_ptr` using `PortableServer::POA::_narrow()`.

   Because `tmp_ref` is of `CORBA::Object` type, which is 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 `_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 69).

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()` is defined as follows:

```
//C++
PortableServer::POA_ptr
create_simple_poa(
    const char*                    poa_name,
    PortableServer::POA_ptr        parent_poa,
    PortableServer::POAManager_ptr poa_manager
)
{
    // Create a policy list.
    // Policies not set in the list get default values.
    //
    CORBA::PolicyList policies;
    policies.length(1);
    int i = 0;
    // Make the POA single threaded.
    //
    policies[i++] = parent_poa->create_thread_policy(
        PortableServer::SINGLE_THREAD_MODEL
    );
    assert(i==1);

    return parent_poa->create_POA(
                          poa_name,
```

```
                                poa_manager,
                                policies);
}
```

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 228.

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

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:

```
// C++
#include <BuildingImpl.h>
...
// Note: PortableServer::Servant is a pointer type - it's
// actually a typedef for PortableServer::ServantBase*.
//
PortableServer::Servant the_Building = 0;
...
the_Building = 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

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:

```
// C++
#include <BuildingImpl.h>
...
CORBA::Object_var tmp_ref;
...
PortableServer::ObjectId_var 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 `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 202; for information about activating objects on demand, see Chapter 11 on page 249.

Export Object References

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()`:

```
//C++
...
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:

```
//C++
void
write_reference(
    CORBA::Object_ptr ref, const char* objref_file
)
{
    CORBA::String_var stringified_ref =
```

```
        global_orb->object_to_string(ref);
    cout << "Writing stringified object reference to "
         << objref_file << endl;

    ofstream os(objref_file);
    os << stringified_ref;
    if (!os.good())
    {
        cerr << "Failed to write to " << objref_file << endl;
    }
}
```

The `ref` object reference is converted to a string, of type `char *` by passing `ref` as an argument to `CORBA::ORB::object_to_string()`. The string is then written to the `objref_file` file.

Note that a smart pointer of `CORBA::String_var` type is used to reference the stringified object reference. The smart pointer automatically deletes the string when it goes out of scope, thereby avoiding a memory leak.

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.

Put the ORB into an Active State

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

```
//C++
...
// Activate the POA Manager and let the ORB process requests.
//
1 root_poa_manager->activate();
2 global_orb->run();
```

The code can be explained as follows:

1. A POA manager can be in four different states: *active*, *holding*, *discarding*, and *inactive*. A POA can accept object requests only after its manager is activated by calling `PortableServer::POAManager::activate()`.

2. `CORBA::ORB::run()` puts the ORB into a state where it listens for client connection attempts and accepts request messages from existing client connections.

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

Shut Down the ORB

The shutdown mechanism for the demonstration application uses Orbix's own `IT_TerminationHandler` class, which enables server applications to handle delivery of `CTRL-C` and similar events in a portable manner (see page 62 and "Termination Handler" on page 219).

Before shutdown is initiated, the server is blocked in the execution of `CORBA::ORB::run()`.

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.

The Orbix termination handler can handle a number of signals or events (see "Create a Termination Handler Object" on page 62). As soon as the server receives one of these signals or events, a thread started by Orbix executes the registered termination handler callback, `termination_handler_callback()`.

The termination handler function is defined as follows:

```
//C++
static void
termination_handler_callback(
   long signal
)
{
1     if (!CORBA::is_nil(orb))
      {
2         global_orb->shutdown(IT_FALSE);
```

```
    }
}
```

The code executes as follows:

1. A check is made to ensure that the `global_orb` variable is initialized.
2. `CORBA::ORB::shutdown()` is invoked. It takes a single boolean argument, the `wait_for_completion` flag.

   When `shutdown()` is called with its `wait_for_completion` flag set to false, a background thread is created to handle shutdown and the call returns immediately. See "Explicit Event Handling" on page 218.

As soon as `termination_handler()` returns, the operating system returns to the prior execution point and the server resumes processing in `CORBA::ORB::run()`.

Server execution now reverts to `main()`:

```
//C++
...
1 global_orb->run();
// Delete the servants.
2 delete the_Building;

// Destroy the ORB and reclaim resources.
try
{
3     global_orb->destroy();
}
catch (...)
{
    // Do nothing.
}
return exit_status;
```

The code executes as follows:

1. After the termination handler completes shutdown, `CORBA::ORB::run()` unblocks and returns.
2. The `BuildingImpl` servant must be explicitly deleted because it is not referenced by a smart pointer.
3. `CORBA::ORB::destroy()` destroys the ORB object.

Note: The `shutdown()` function is not called after `CORBA::ORB::run()` returns, because `shutdown()` is already called in the signal handler. It is illegal to call `shutdown()` more than once on the same ORB object.

Complete Source Code for server.cxx

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

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

#include "it_random_funcs.h"
#include <iostream.h>
#include <fstream.h>
#include <string.h>
#include <stdlib.h>
#include <it_ts/termination_handler.h>
#include <omg/PortableServer.hh>
#include "BuildingImpl.h"

// global_orb -- make ORB global so all code can find it.
//
CORBA::ORB_var
global_orb = CORBA::ORB::_nil();

// termination handler callback handles Ctrl-C-like signals/events
// by shutting down the ORB. This causes ORB::run() to return,
//  and allows the server to shut down gracefully.

static void
termination_handler_callback(
   long signal
)
```

```
{

    cout << "Processing shutdown signal " << signal << endl;
    if (!CORBA::is_nil(orb))
    {
        cout << "ORB shutdown ... " << flush;
        orb->shutdown(IT_FALSE);
        cout << "done." << endl;
    }
}

// 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.
//
void
write_reference(
    CORBA::Object_ptr               ref,
    const char*                     objref_file
)
{
    CORBA::String_var stringified_ref =
        global_orb->object_to_string(ref);
    cout << "Writing stringified object reference to "
         << objref_file << endl;

    ofstream os(objref_file);
    os << stringified_ref;
    if (!os.good())
    {
        cerr << "Failed to write to " << objref_file << endl;
    }
}

// create_simple_poa() -- Create a POA for simple servant
   management.
//
PortableServer::POA_ptr
create_simple_poa(
    const char*                   poa_name,
    PortableServer::POA_ptr       parent_poa,
```

```
    PortableServer::POAManager_ptr poa_manager
)
{
    // Create a policy list.
    // Policies not set in the list get default values.
    //
    CORBA::PolicyList policies;
    policies.length(1);
    int i = 0;
    // Make the POA single threaded.
    //
    policies[i++] = parent_poa->create_thread_policy(
        PortableServer::SINGLE_THREAD_MODEL
    );
    assert(i==1);

    return parent_poa->create_POA(poa_name,
                                  poa_manager,
                                  policies);
}

// main() -- set up a POA, create and export object references.
//
int
main(int argc, char **argv)
{
    int exit_status = 0;             // Return code from main().

    // Instantiate termination handler
    IT_TerminationHandler
    termination_handler(termination_handler_callback);

    // Variables to hold our servants.
    // Note: PortableServer::Servant is a pointer type - it's
    // actually a typedef for PortableServer::ServantBase*.
    //
    PortableServer::Servant the_Building = 0;

    try
    {
        // For temporary object references.
        CORBA::Object_var tmp_ref;
```

```
    // Initialise the ORB and Root POA.
    //
    cout << "Initializing the ORB" << endl;
    global_orb = CORBA::ORB_init(argc, argv);
    tmp_ref =
        global_orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var root_poa =
        PortableServer::POA::_narrow(tmp_ref);
    assert(!CORBA::is_nil(root_poa));
    PortableServer::POAManager_var root_poa_manager
        = root_poa->the_POAManager();
    assert(!CORBA::is_nil(root_poa_manager));

    // Now create our own POA.
    //
    PortableServer::POA_var 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.
    //
    PortableServer::ObjectId_var oid;

    // Create a servant for interface Building.
    //
    the_Building = 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");

    // Activate the POA Manager and let the ORB process
    // requests.
    //
    root_poa_manager->activate();
    cout << "Waiting for requests..." << endl;
    global_orb->run();
}
catch (CORBA::Exception& e)
{
    cout << "Unexpected CORBA exception: " << e << endl;
```

```
            exit_status = 1;
        }
        // Delete the servants.
        //
        delete the_Building;

        // Destroy the ORB and reclaim resources.
        //
        try
        {
            global_orb->destroy();
        }
        catch (...)
        {
            // Do nothing.
        }
        return exit_status;
    }
```

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.

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 definition typically has the following components:

- Operation definitions
- Attribute definitions
- Exception definitions
- Type definitions
- Constant definitions

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 91).

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.

A client can also issue non-blocking, or asynchronous, invocations. For more information, see Chapter 12 on page 267.

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 information about exception handling, see Chapter 13 on page 277.

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 16 shows the inheritance hierarchy for this interface.

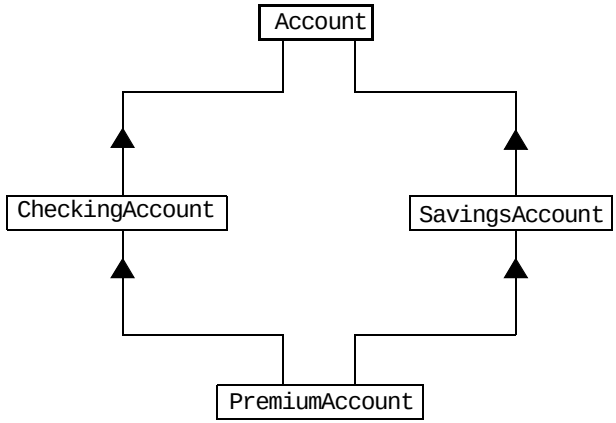

Figure 16: *Multiple inheritance of IDL interfaces*

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 a `CORBA::MARSHAL` exception.
- 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 a `CORBA::NO_IMPLEMENT` exception.
- 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 2:

Table 2: *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 C++ abstract base classes. 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 types such as `enum` and `struct`, and `string`
- Pseudo objects

Built-in Types

Table 3 lists built-in IDL types.

Table 3: *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 15 on page 303.

Extended Built-in Types

Table 4 lists extended built-in IDL types.

Table 4: *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. Because Orbix currently does not support character set negotiation, use this type only for applications that are distributed across the same platform.

wstring

Type `wstring` is the wide-character equivalent of type `string` (see page 94). 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`.

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 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 inaccurately represent decimal fractions 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 101).

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 102) 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 102) 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 code 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.

Integer Constants

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 Constants

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 Constants

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 Constants

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 Constants

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 Constants

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 Constants

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 Constants

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.

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.

Precedence

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

5 Developing Applications with Genies

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

Two genies generate code that you can use immediately for application development:

- `cpp_poa_genie.tcl` reads IDL code and generates C++ source files that you can compile into a working application.
- `cpp_poa_op.tcl` generates the C++ signatures of specified operations and attributes and writes them to a file. You can use this genie on new or changed interfaces, then update existing source code with the generated signatures.

Starting Development Projects

The C++ genie `cpp_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 makefile, 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, and to manage memory for various IDL data types.

Genie Syntax

`cpp_poa_genie.tcl` uses the following syntax:

`idlgen cpp_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 5:

Table 5: *Component specifier arguments to cpp_poa_genie.tcl*

Component specifier	Output
`-all`	All components: server, servant, client, and makefile (see page 111).
`-servant`	Servant classes to implement the selected interfaces (see page 114).
`-server`	Server main program (see page 118)
`-client`	Client main program (see page 121).
`-makefile`	A makefile to compile server and client applications (see page 122).

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 6:

Table 6: *Optional switches to cpp_poa_genie.tcl*

Option	Description
`-complete/-incomplete`	Controls the completeness of the code that is generated for the specified components (see page 122).
`-dir`	Specifies where to generate file output (see page 126).
`-include`	Specifies to generate code for included files (see page 113).
`-interface-spec`	Specifies to generate code only for the specified interfaces (see page 112).
`-v/-s`	Controls the level of verbosity (see page 126).

Specifying Application Components

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

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

bankdemo.idl:
idlgen: creating BankDemo_BankImpl.h
idlgen: creating BankDemo_BankImpl.cxx
idlgen: creating BankDemo_AccountImpl.h
idlgen: creating BankDemo_AccountImpl.cxx
idlgen: creating server.cxx
idlgen: creating client.cxx
idlgen: creating call_funcs.h
idlgen: creating call_funcs.cxx
idlgen: creating it_print_funcs.h
idlgen: creating it_print_funcs.cxx
```

```
idlgen: creating it_random_funcs.h
idlgen: creating it_random_funcs.cxx
idlgen: creating Makefile
```

Alternatively, you can use cpp_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 servant:

```
> idlgen cpp_poa_genie.tcl -servant bankdemo.idl

bankdemo.idl:
idlgen: creating BankDemo_BankImpl.h
idlgen: creating BankDemo_BankImpl.cxx
idlgen: creating BankDemo_AccountImpl.h
idlgen: creating BankDemo_AccountImpl.cxx
idlgen: creating it_print_funcs.h
idlgen: creating it_print_funcs.cxx
idlgen: creating it_random_funcs.h
idlgen: creating it_random_funcs.cxx
```

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

Selecting Interfaces

By default, cpp_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 cpp_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 cpp_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 cpp_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 7 contains some common wildcard patterns:

Table 7: *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 cpp_poa_genie.tcl -all -include bankdemo.idl
```

The default for this option is set in the configuration file through `default.cpp_poa_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 cpp_poa_genie.tcl -servant bankdemo.idl
```

The genie constructs the implementation class name from the scoped name of the interface, replacing double colons (`::`) with an underscore (_) and adding a suffix—by default, `Impl`.. The default suffix is set in the configuration file through `default.cpp.impl_class_suffix`.

For example, `BankDemo::Account` is implemented by class `BankDemo_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 8, that let you control how servant classes are generated:

Table 8: *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.

Table 8: *Arguments that control servant generation*

Argument	Purpose
`-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 237.
`-refcount` `-norefcount`	Choose whether or not servants are reference counted.

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 122.

-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.cpp_poa_genie.want_tie`.

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

```
class BankDemo_AccountImpl : public virtual POA_BankDemo::Account
```

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

```
// C++
POA_BankDemo::Account*
BankDemo_AccountImpl::_create(PortableServer::POA_ptr the_poa)
{
    return new BankDemo_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:

```
// C++
POA_BankDemo::Account*
BankDemo_AccountImpl::_create(PortableServer::POA_ptr the_poa)
{
1       BankDemo_AccountImpl* tied_object =
            new BankDemo_AccountImpl();
2       POA_BankDemo::Account* the_tie =
            new POA_BankDemo_Account_tie<BankDemo_AccountImpl>(
                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.cpp_poa_genie.want_inherit`.

-default_poa

In the standard CORBA C++ 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 `cpp_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 236.

-refcount/-norefcount

Multi-threaded servers need to reference-count their servants in order to avoid destroying a servant on one thread that is still in use on another. The POA specification provides the standard functions `_add_ref()` and `_remove_ref()` to support reference counting, but by default they do nothing.

- `-refcount` generates servants that inherit from the standard class `PortableServer::RefCountServantBase`, which enables reference counting. For example:

  ```
  class BankDemo_AccountImpl
   : public virtual POA_BankDemo::Account,
     public virtual PortableServer::RefCountServantBase
  ```

- `-norefcount` specifies that servants do not inherit from `RefCountServantBase`.

The `-refcount` option is automatically enabled if you use the `-threads` option (see page 119).

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

Note: `-refcount` is invalid with `-tie`. The genie issues a warning if you combine these options. Tie templates as defined in the POA standard do not support reference counting, and the genie cannot change their inheritance. It is recommended that you do not use the tie approach for multi-threaded servers.

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 `server.cxx` that contains a `main` program:

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

The server program performs the following steps:

1. Initializes the ORB and POA.
2. Installs a signal handler to shut down gracefully if the server is killed via SIGTERM on Unix or a CTRL-C event on Windows.
3. 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`.
4. Catches any exceptions and print a message.

The -server option can take one or more arguments, shown in Table 9, that let you modify server behavior:

Table 9: *Options affecting the server*

Command line option	Purpose
-threads -nothreads	Choose a single or multi-threaded server. The -threads argument also implies -refcount (see page 117).
-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

The -nothreads option 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.

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

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

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.

-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 229). 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. After the ORB is shut down, `main()` deletes the servants.

  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 251). 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. The servant activator deletes the servants when the POA is destroyed.

  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 256). 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, and deletes it when the operation is complete.

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 264.

-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 18 on page 377.

- `-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.cpp_poa_genie`.

Implementing a Client

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

```
> idlgen cpp_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 a Makefile

The `-makefile` option generates a makefile that can build the server and client applications. The makefile provides the following targets

- `all`: Compile and link the client and server.
- `clean`: Delete files created during compile and link.
- `clean_all`: Like clean, it also deletes all the source files generated by idlgen, including the makefile itself.

To build the client and server, enter `nmake` (Windows) or `make` (UNIX).

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.cpp_poa_genie.want_complete`.

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

```
> idlgen cpp_poa_genie.tcl -servant -complete bankdemo.idl
> idlgen cpp_poa_genie.tcl -client -complete bankdemo.idl
> idlgen cpp_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, throw and catch exceptions, and perform memory management.

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

Setting `-complete servant` and `-incomplete servant` yields the required source files for each IDL interface. Either option generate the following files for interface `Account`:

```
BankDemo_AccountImpl.h
BankDemo_AccountImpl.cxx
```

Incomplete Servant

The `-incomplete` option specifies to generate servant class `BankDemo_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 `deposit()` operation:

```
void
BankDemo_AccountImpl::deposit(
    BankDemo::CashAmount amount
) throw(
    CORBA::SystemException
)
{
        throw 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 several files that provide the functionality required to generate random values for parameter passing, and to print those values:

```
it_print_funcs.h
it_print_funcs.cxx
it_random_funcs.h
it_random_funcs.cxx
```

Member methods are fully implemented to print parameter values and, if required, return a value to the client. For example, the following code is generated for the `deposit()` operation:

```
void
BankDemo_AccountImpl::deposit(
    BankDemo::CashAmount amount
) throw(
    CORBA::SystemException
)
{
    // Diagnostics: print the values of "in" and "inout" parameters
    cout << "BankDemo_AccountImpl::deposit(): "
         << "called with..."
         << endl;
    cout << "\tamount = ";
    IT_print_BankDemo_CashAmount(cout, amount, 3);
    cout << endl;

    // Diagnostics.
    cout << "BankDemo_AccountImpl::deposit(): returning"
         << endl;

}
```

Client Code

In a completely implemented client, `cpp_poa_genie.tcl` generates the client source file `call_funcs.cxx`, which contains method calls that invoke on all operation and attributes of each 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 `it_random_funcs.cxx`, and utility methods to print the values of IDL types are generated in the file `it_print_funcs.cxx`.

An incomplete client contains no invocations.

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

General Options

You can supply switches that control cpp_poa_genie.tcl genie output:

-dir: By default, cpp_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, cpp_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 makefile.
2. In separate windows, run first the server, then the client applications.

Generating Signatures of Individual Operations

IDL interfaces sometimes change during development. A new operation might be added to an interface, or the signature of an existing operation might change. When such a change occurs, you must update existing C++ code with the signatures of the new or modified operations. You can avoid much of this work with the cpp_poa_op.tcl genie. This genie prints the C++ signatures of specified operations and attributes to a file. You can then paste these operations back into the application source files.

For example, you might add a new operation close() to interface BankDemo::Account. To generate the new operation, run the cpp_poa_op.tcl genie:

```
> idlgen cpp_poa_op.tcl bankdemo.idl "*::close"

idlgen: creating tmp
Generating signatures for BankDemo::Account::close
```

As in this example, you can use wildcards to specify the names of operations or attributes. If you do not explicitly specify any operations or attributes, the genie generates signatures for all operations and attributes.

By default, wild cards are matched only against names of operations and attributes in the specified IDL file. If you specify the `-include` option, wildcards are also matched against all operations and attributes in the included IDL files.

By default, `cpp_poa_op.tcl` writes generated operations to file `tmp`. You can specify a different file name with the `-o` command-line option:

```
> idlgen cpp_poa_op.tcl bankdemo.idl -o ops.txt "*::close"

bankdemo.idl:
idlgen: creating ops.txt
Generating signatures for BankDemo::Account::close
```

Configuration Settings

The configuration file `idlgen.cfg` contains default settings for the C++ genie `cpp_poa_genie.tcl` at the scope `default.cpp_poa_genie`.

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

For a full listing of these settings, refer to the *Orbix 2000 Code Generation Toolkit Programmer's Guide*.

6 ORB Intialization and Shutdown

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

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

- Initialize the ORB by calling `CORBA::ORB_init()`.
- Shut down and destroy the ORB at the end of `main()`, by calling `shutdown()` and `destroy()` on the ORB.

Orbix also provides its own `IT_TerminationHandler` class, which enables applications to handle delivery of `Ctrl-C` and similar events in a portable manner. For more information, see "Termination Handler" on page 219

Initializing the ORB Runtime

Before an application can start any CORBA-related activity, it must initialize the ORB runtime by calling `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. `ORB_init()` scans its arguments parameter for command-line options that start with `-ORB` and removes them. The arguments that remain can be assumed to be application-specific.

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 *Orbix 2000 Administrator's Guide*.

C++ Mapping

`ORB_init()` is defined as follows:

```
namespace CORBA {

// ...
    ORB_ptr ORB_init(
                int &           argc,
                char **         aaccv,
                const char *    orb_identifier = ""
            );
    // ...
}
```

`ORB_init()` expects a reference to `argc` and a non-constant pointer to `aaccv`. `ORB_init()` scans the passed argument vector for command-line options that start with `-ORB` and removes them.

Registering Portable Interceptors

During ORB intialization, portable interceptors are instantiated and registered through an ORB intializer. The client and server applications must register the ORB initializer before calling `ORB_init()`. For more information, see "Registering Portable Interceptors" on page 529.

Shutting Down the ORB

For maximum portability and to ensure against resource leaks, a client or server should 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.

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.

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 145, and "Managing Server Objects" on page 221.

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 13 on page 226). 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`.

```
CORBA::PolicyList policies;
policies.length (2);

// Use root POA to create POA policies
policies[0] = poa->create_lifespan_policy
    (PortableServer::PERSISTENT)
policies[1] = poa->create_id_assignment_policy
    (PortableServer::USER_ID)
```

Orbix also provides several proprietary policies to control POA behavior (see page 226). 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:

```
#include <omg/messaging.hh>;
// ...
CORBA::PolicyList policies(1);
policies.length(1);
CORBA::Any policy_value;
policy_any <<= Messaging::SYNC_WITH_SERVER;

policies[0] = orb->create_policy(
    Messaging::SYNC_SCOPE_POLICY_TYPE, 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 141).

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 *Orbix 2000 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 135).
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 228).

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
CORBA::Object_var obj =
    orb->resolve_initial_references( "RootPOA" );
PortableServer::POA_var poa = POA::_narrow( obj );

//create policy object
```

```
CORBA::PolicyList policies;
policies.length (2);

// set policy object with desired policies
policies[0] = poa->create_lifespan_policy
    (PortableServer::PERSISTENT)
policies[1] = poa->create_id_assignment_policy
    (PortableServer::USER_ID)

//create a POA for persistent objects
poa = poa->create_POA( "persistentPOA", NULL, 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 228.

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:

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 *Orbix 2000 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 135).
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 135).

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 139).

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);
};
```

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 141).

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)

    interface RebindPolicy : CORBA::Policy {
            readonly attribute RebindMode rebind_mode;
    };


    // 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:

```
#include <omg/messaging.hh>
...
// get reference to PolicyManager

CORBA::Object_var object;
object = orb->resolve_initial_references("ORBPolicyManager");

// narrow
CORBA::PolicyManager_var policy_mgr =
    CORBA::PolicyManager::_narrow(object);

// set SyncScope policy at ORB scope (not shown)
// ...

// get SyncScope policy at ORB scope
CORBA::PolicyTypeSeq types;
types.length(1);
types[0] = SYNC_SCOPE_POLICY_TYPE;

// get PolicyList from ORB's PolicyManager
CORBA::PolicyList_var pList =
    policy_mgr->get_policy_overrides( types );

// evaluate first Policy in PolicyList
Messaging::SyncScopePolicy_var sync_p =
    Messaging::SyncScopePolicy::_narrow( pList[0] );

Messaging::SyncScope sync_policy = sync_p->synchronization();

cout << "Effective SyncScope policy at ORB level is "
     << sync_policy << endl;
```

8 Developing a Client

A CORBA client initializes the ORB runtime, handles object references, invokes operations on objects, and handles exceptions that these operations throw.

This chapter covers the following topics:

- Mapping of IDL interfaces to proxies and references.
- Handling object reference types.
- Initializing the ORB runtime.
- Invoking operations and parameter passing rules.
- Using quality of service policies.

For information about exception handling, see Chapter 13.

Interfaces and Proxies

When you compile IDL, the compiler maps each IDL interface to a client-side *proxy* class of the same name. 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 proxy class definition for the client implementation:

```
namespace BankDemo
{
    typedef CORBA::Float CashAmount;
    // ...

    class Account : public virtual CORBA::Object
    {
        // ...
        virtual void withdraw( CashAmount amount ) = 0;
    }
    // other operations 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 `CORBA::Object`, so the client can access all the inherited functionality of a CORBA object.
- `Account::withdrawal` and all other member methods are defined as pure virtual, so the client code cannot instantiate the Account proxy class or any other proxy class. Instead, clients can access the `Account` object only indirectly through object references.

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. For each generated proxy class, the IDL compiler also generates two other classes: *interface*_var and *interface*_ptr, where *interface* is the name of the proxy class. Briefly, _ptr types are unmanaged reference types, while _var types can be characterized as smart pointers.

Both reference types support the indirection operator ->; when you invoke an operation on a _var or _ptr reference, the corresponding proxy object redirects the C++ call across the network to the appropriate member method of the object's servant.

While _ptr and _var references differ in a number of ways, they both act as handles to the corresponding client proxy. The client code only needs to obtain an object reference and use it to initialize the correct _ptr or _var reference. The underlying proxy code and ORB runtime take all responsibility for ensuring transparent access to the server object

For example, given the previous IDL, the IDL compiler generates two object reference types to the CORBA object, Bank::Account: Account_ptr and Account_var. You can use either reference type to invoke operations such as withdrawal() on the Bank::Account object. Thus, the following two invocations are equivalent:

```
// ...
// withdraw_amt is already initialized

// Use a _ptr reference
Account_ptr accp = ...;     // get reference...
balance = accp->withdrawal( withdraw_amt );

// Use a _var reference
Account_var accv = ...;     // get reference...
balance = accv->withdrawal( withdraw_amt );
```

Note: Because _ptr types are not always implemented as actual C++ pointers, you should always use the _ptr definition. Regardless of the underlying mapping, a _ptr type is always guaranteed to behave like a pointer, so it is portable across all platforms and language mappings.

Counting References

When you initialize a _var or _ptr reference with an object reference for the first time, the client instantiates a proxy and sets that proxy's reference count to one. Each proxy class has a _duplicate() method, which allows a client to create a copy of the target proxy. In practice, this method simply increments the reference count on that proxy and returns a new _ptr reference to it. Actual methods for copying _ptr and _var references differ and are addressed separately in this chapter; conceptually, however, the result is the same.

For example, given an object reference to the Account interface, the following client code initializes a _ptr reference as follows:

```
Account_ptr accp1 = ...; // get reference somehow
```

This instantiates an Account object proxy and automatically sets its reference count to one:

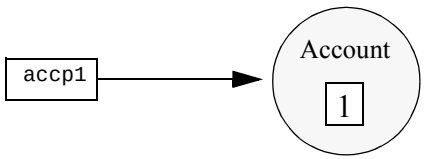

Figure 17: *Reference count for Account proxy is set to one.*

The following code copies accp1 into reference accp2, thus incrementing the Account proxy's reference count to 2

```
Account_ptr accp2 = Account::_duplicate(accp1);
```

The client now has two initialized _ptr references, accp1 and accp2. Both refer to the same proxy, so invocations on either are treated as invocations on the same object.

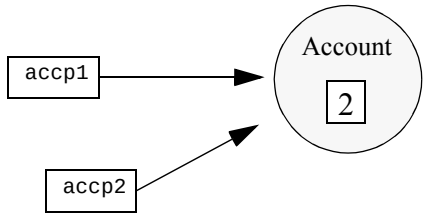

Figure 18: *Reference for Account proxy is incremented to 2.*

When you release a reference, the reference count of the corresponding proxy is automatically decremented. When the reference count is zero, Orbix deallocates the proxy. You can release references in any order, but you can only release a reference once, and you must not use any reference after it is released.

Note: A server object is completely unaware of its corresponding client proxy and its life cycle. Thus, calling `release()` and `_duplicate()` on a proxy reference has no effect on the server object.

Nil References

Nil references are analogous to C++ null pointers and contain a special value to indicate that the reference points nowhere. Nil references are useful mainly to indicate “not there” or optional semantics. For example, if you have a lookup operation that searches for objects via a key, it can return a nil reference to indicate the “not found” condition instead of raising an exception. Similarly, clients can pass a nil reference to an operation to indicate that no reference was passed for this operation—that is, you can use a nil reference to simulate an optional reference parameter.

You should only use the `CORBA::is_nil()` method to test whether a reference is nil. All other attempts to test for nil have undefined behavior. For example, the following code is not CORBA-compliant and can yield unpredictable results:

```
Object_ptr ref = ...;
if (ref != 0) {         // WRONG! Use CORBA::is_nil
    // Use reference...
}
```

You cannot invoke operations on a nil reference. For example, the following code has undefined behavior:

```
Account_ptr accp = Account::_nil();
// ...
CORBA::CashAmount bal = accp->balance();  // Crash imminent!
```

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);
        // ...
    }
};
```

In C++, these operations are mapped to `CORBA::Object` member methods as follows:

```
// In namespace CORBA:

class Object {
public:
```

```
    static Object_ptr _duplicate(Object_ptr obj);
    void release(Type_ptr);
    Boolean is_nil(Type_ptr p);
    Boolean _is_a(const char * repository_id);
    Boolean _non_existent();
    Boolean _is_equivalent(Object_ptr other_obj);
    ULong   _hash(ULong max);
    // ...
};
```

The `is_nil()` method is discussed earlier in this chapter (see page 149). Methods `_duplicate()`, and `release()` are discussed later in this chapter (see page 153). This section covers the remaining methods.

_is_a

The `_is_a()` method is similar to `_narrow()` in that it lets you to determine whether an object supports a specific interface. For example:

```
CORBA::Object_ptr obj = ...;    // Get a reference

if (!CORBA::is_nil(obj) && obj->_is_a("IDL:BankDemo/Account:1.0"))
    // It's an Account object...
else
    // Some other type of object...
```

The test for nil in this code example prevents the client program from making a call via a nil 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 `_narrow()` to ascertain whether an object supports the desired interface.

_non_existent

The `_non_existent()` method 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

The _is_equivalent() method 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

The _hash() method 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.

Using _ptr References

The IDL compiler defines a _ptr reference type for each IDL interface. In general, you can think of a _ptr reference as a pointer to a proxy instance, with the same semantics and requirements as any C++ pointer.

Duplicating and Releasing References

To make a copy of a _ptr reference, invoke the static _duplicate() member method on an existing object reference. For example:

```
Account_ptr acc1 = ...;             // Get ref from somewhere...
Account_ptr acc2;                   // acc2 has undefined contents
acc2 = Account::_duplicate(acc1);   // Both reference same Account
```

_duplicate() makes an exact copy of a reference. The copy and the original are indistinguishable from each other. As shown earlier (see “Counting References” on page 148), _duplicate() also makes a deep copy of the target reference, so the reference count on the proxy object is incremented. Consequently, you must call release() on all duplicated references to destroy them and prevent memory leaks.

To destroy a reference, use the release method. For example:

```
Account_ptr accp = ...; // Get reference from somewhere...
// ...Use accp
CORBA::release(accp);     // Don't want to use Account anymore
```

_duplicate() is type safe. To copy an Account reference, supply an Account reference argument to _duplicate(). Conversely, the CORBA namespace contains only one release() method, which releases object references of any type.

Widening and Narrowing _ptr References

Proxy classes emulate the inheritance hierarchy of the IDL interfaces from which they are generated. Thus, you can widen and narrow _ptr references to the corresponding proxies.

Widening Assignments

Object references to proxy instances conform to C++ rules for type compatibility. Thus, you can assign a derived reference to a base reference, or pass a derived reference where a base reference is expected.

For example, the following IDL defines the `CheckingAccount` interface, which inherits from the `Account` interface shown earlier:

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

Given this inheritance hierarchy, the following widening assignments are legal:

```
CheckingAccount_ptr ck = ...;    // Get checking account reference
Account_ptr accp = ck;           // Widening assignment
CORBA::Object_ptr obj1 = ck;     // Widening assignment
CORBA::Object_ptr obj2 = accp;   // Widening assignment
```

Note: Because all proxies inherit from `CORBA::Object`, you can assign any type of object reference to `Object_ptr`, such as `_ptr` references `obj1` and `obj2`.

Ordinary assignments between _ptr references have no effect on the reference count. Thus, the assignments shown in the previous code can be characterized as shown in Figure 19:

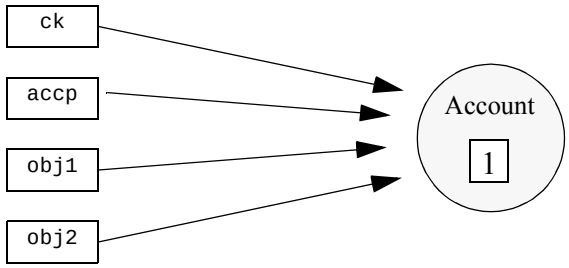

Figure 19: *Multiple _ptr references to a proxy object can leave the reference count unchanged.*

Because the reference count is only 1, calling `release()` on any of these references decrements the proxy reference count to 0, causing Orbix to deallocate the proxy. Thereafter, all references to this proxy are invalid.

Type-Safe Narrowing of _ptr References

For each interface, the IDL compiler generates a static `_narrow()` method that lets you down-cast a `_ptr` reference at runtime. For example, the following code narrows an `Account` reference to a `CheckingAccount` reference:

```
BankDemo::Account_ptr accp = ..; // get a reference from somewhere
BankDemo::CheckingAccount_ptr ckp =
    BankDemo::CheckingAccount::_narrow(accp);
if (CORBA::is_nil(ckp))
{
    // accp is not of type CheckingAccount
}
else
{
    // accp is a CheckingAccount type, so ckp is a valid reference
}
// ...
```

```
// release references to Account proxy
CORBA::release(ckp);
CORBA::release(accp);
```

Because _narrow() calls _duplicate(), it increments the reference count on the Account proxy—in this example, to 2. Consequently, the code must release both references.

Using _var References

The IDL compiler defines a _var class type for each IDL interface, which lets you instantiate _var references in the client code. Each _var references takes ownership of the reference that it is initialized with, and calls CORBA::release() when it goes out of scope.

If you initialize a _var reference with a _ptr reference, you cannot suffer a resource leak because, when it goes out of scope, the _var reference automatically decrements the reference count on the proxy.

_var references are also useful for gaining exception safety. For example, if you keep a reference you have just obtained as a _var reference, you can throw an exception at any time and it does not leak the reference because the C++ run time system calls the _var's destructor as it unwinds the stack

_var Class Member Methods

Given the Account interface shown earlier, the IDL compiler generates an Account_var class with the following definition:

```
class Account_var{
public:
    Account_var();
    Account_var(Account_ptr &);
    Account_var(const Account_var &);
    ~Account_var();
    Account_var & operator=(Account_ptr &);
    Account_var & operator=(const Account_var &);
    operator Account_ptr & ();
    Account_ptr in() const;
    Account_ptr & in inout();
    Account_ptr & in out();
    Account_ptr _retn();
```

```
private:
    Account_ptr p; //actual reference stored here
};
```

Account_var()

The default constructor initializes the private _ptr reference to nil.

Account_var(Account_ptr &)

Constructing a _var from a _ptr reference passes ownership of the _ptr reference to the _var. This method leaves the proxy reference count unchanged.

Account_var(const Account_var &)

Copy-constructing a _var makes a deep copy by calling _duplicate() on the source reference. This method increments the proxy reference count.

~Account_var()

The destructor decrements the proxy reference count by calling release().

Account_var & operator=(Account_ptr &)
Account_var & operator=(const Account_var &)

Assignment from a pointer passes ownership and leaves the proxy reference count unchanged; assignment from another Account_var makes a deep copy and increments the reference count.

operator Account_ptr &()

This conversion operator lets you pass a _var reference where a _ptr reference is expected, so use of _var references is transparent for assignment and parameter passing.

Account_ptr operator->() const

The indirection operator permits access to the member methods on the proxy via a _var by returning the internal _ptr reference.

Account_ptr in() const
Account_ptr & inout()
Account_ptr & out()

Explicit conversion operators are provided for compilers that incorrectly apply C++ argument-matching rules.

Account_ptr _retn()

The `_retn()` method removes ownership of a reference from a `_var` without decrementing the reference count. This is useful if a method must allocate and return a `_var` reference, but also throws exceptions.

Widening and Narrowing _var References

You can copy-construct and assign from `_var` references, but only if both references are of the same type. For example, the following code is valid:

```
Account_var accv1 = ...;    // get object reference
Account_var accv2(accv1);   // Fine, deep copy
accv1 = accv2;              // Fine, deep assignment
```

Unlike `_ptr` references, `_var` references have no inheritance relationship, so implicit widening among `_var` references is not allowed. For example, you cannot use a `CheckingAccount_var` to initialize an `Account_var`:

```
CheckingAccount_var ckv = ...; // get object reference
accv1 = ckv;                   // Compile-time error
Account_var accv3(ckv);        // Compile-time error
```

To widen a `_var` reference, you must first call `_duplicate()` on the original `_var`. Although `_duplicate()` expects a `_ptr` reference, a `_var` can be supplied in its place, as with any method that expects a `_ptr` reference. `_duplicate()` returns a `_ptr` reference that can then be implicitly widened.

For example, in the following statement, `_duplicate()` receives a `CheckingAccount_var`:

```
Account_var accv1(CheckingAccount::_duplicate(ckv));
```

`_duplicate()` returns a `CheckingAccount_ptr` that is implicitly widened to an `Account_ptr` as the argument to the `Account_var` constructor. The constructor in turn takes ownership, so the copy made by `_duplicate()` is not leaked.

In the next statement, `_duplicate()` expects an `Account_ptr`:

```
Account_var accv2(Account::_duplicate(ckv));
```

In fact, a `CheckingAccount_var` argument is supplied, which has a conversion operator to `CheckingAccount_ptr`. A `CheckingAccount_ptr` can be passed where an `Account_ptr` is expected, so the compiler finds an argument match. `_duplicate()` makes a copy of the passed reference and returns it as an `Account_ptr`, which is adopted by the `Account_var`, and no leak occurs.

You can also use `_duplicate()` for implicit `_var` widening through assignment, as in these examples:

```
accv1 = CheckingAccount::_duplicate(ckv);
accv2 = Account::_duplicate(ckv);
```

You can freely mix `_ptr` and `_var` references; you only need to remember that when you give a `_ptr` reference to a `_var` reference, the `_var` takes ownership:

```
// Be careful of ownership when mixing _var and _ptr:
{
  CheckingAccount_var ckv = ...; // Get reference...
  Account_ptr accp = ckv;        // OK, but ckv still has ownership

  // Can use both ckv and accp here...

  CheckingAccount_ptr ckp = ...; // Get reference...
  ckv = ckp;                     // ckv now owner, accp dangles

  level = accp->balance();       // ERROR - accp dangles
} // ckv automatically releases its reference, ckp dangles!
level = ckp->balance()           // ERROR -ckp dangles
```

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.

The `object_to_string()` and `string_to_object()` operations are defined in C++ as follows:

```
// In <corba/orb.hh>:
namespace CORBA {
    // ...
    class ORB {
    public:
        char *     object_to_string(Object_ptr op);
        Object_ptr string_to_object(const char *);
        // ...
    };
    // ...
}
```

For example, the following code stringifies an `Account` object reference:

```
BankDemo::Account_ptr accp = ...; // Account reference

// Write reference as a string to stdout
//
try {
    CORBA::String_var str = orb->object_to_string(accp);
    cout << str << endl;
} catch (...) {
    // Deal with error...
}
```

The example puts the return value from `object_to_string` in a `String_var`. This ensures that the string is not leaked. This code prints an IOR (interoperable reference) string whose format is similar to this:

```
IOR:
010000002000000049444c3a61636d652e636f6d2f4943532f436f6e74726f6c
c65723a312e3000010000000000000004a000000010102000e0000003139322e3
36382e312e3231300049051b0000003a3e0231310c01000000c7010000234800
0080000000000000000000010000000600000006000000010000001100
```

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.

To convert a string back into a reference, call `string_to_object()`:

```
// Assume stringified reference is in aaccv[1]

try {
    CORBA::Object_ptr obj;
    obj = orb->string_to_object(accv[1]);
    if (CORBA::is_nil(obj))
        throw 0;    // accv[1] is nil

    BankDemo::Account_ptr accp = BankDemo::Account::_narrow(obj);
    if (CORBA::is_nil(accp))
        throw 0;    // Not an Account reference

    // Use accp reference...

    CORBA::release(accp);   // Avoid leak

} catch (...) {
    // 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 nil reference. Nil 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 152).

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 18 on page 377).

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 *Orbix 2000 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 `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 Intialization and Shutdown" on page 129.

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 C++ proxy class Example. A client invokes on these methods as if their implementations existed within its own address space:

```
namespace Test {
// ...
class Example : public virtual CORBA::Object
    {
    public:
    // ...
        virtual char* name() = 0;
        virtual void name(const char*  _itvar_name) = 0;
        virtual void set_address(const char*  addr) = 0;
        virtual char* get_address() = 0;
        // ...
    };
};
```

Passing Parameters in Client Invocations

The C++ mapping has strict rules on passing parameters to operations. Several objectives underlie these rules:

- Avoid data copying.

- Deal with variable-length types, which are allocated by the sender and deallocated by the receiver.
- Map the source code so it is location-transparent; source code does not need to consider whether or not client and server are collocated.

In general, a variable-length parameter is always dynamically allocated, and the receiver of the value is responsible for deallocation. For variable-length `out` parameters and return values, the server allocates the value and the client deallocates it.

For string, reference, and variable-length array `inout` parameters, the client dynamically allocates the value and passes it to the server. The server can either leave the initial value's memory alone or it can deallocate the initial value and allocate a different value to return to the client; either way, responsibility for deallocation of a variable-length `inout` parameter remains with the client.

All other parameters are either fixed-length or `in` parameters. For these, dynamic allocation is unnecessary, and parameters are passed either by value for small types, or by reference for complex types.

Simple Parameters

For simple fixed-length types, parameters are passed by value if they are `in` parameters or return values, and are passed by reference if they are `inout` or `out` parameters.

For example, the following IDL defines an operation with simple parameters:

```
interface Example {
    long op(
            in long in_p, inout long inout_p, out long out_p
    );
};
```

The proxy member method signature is the same as the signature of any other C++ method that passes simple types in these directions:

```
virtual CORBA::Long
op(
    CORBA::Long     in_p,
    CORBA::Long &   inout_p,
    CORBA::Long &   out_p
```

```
) = 0;
```

For example, a client can invoke op as follows:

```
Example_var ev = ...;               // Get reference

CORBA::Long inout = 99;             // Note initialization
CORBA::Long out;                    // No initialization needed
CORBA::Long ret_val;

ret_val = ev->op(500, inout, out);  // Invoke CORBA operation

cout << "ret_val: " << ret_val << endl;
cout << "inout: " << inout << endl;
cout << "out: " << out << endl;
```

The client passes the constant `500` as the `in` parameter. For the `inout` parameter, the client passes the initial value `99`, which the server can change. No initialization is necessary for the `out` parameter and the return value. No dynamic allocation is required; the client can pass variables on the stack, on the heap, or in the data segment (global or static variables).

Fixed-Length Complex Parameters

For fixed-length complex types such as fixed-length structures, parameters are passed by reference or constant reference and are returned by value.

For example, the following IDL defines an operation with fixed-length complex parameters:

```
struct FLS {            // Fixed-Length Structure
    long    long_val;
    double  double_val;
};

interface Example {
    FLS op(in FLS in_p, inout FLS inout_p, out FLS out_p);
};
```

The corresponding proxy method has the following signature:

```
typedef FLS & FLS_out;
// ...
virtual FLS
op(const FLS & in_p, FLS & inout_p, FLS_out out_p) = 0;
```

Using the generated proxy method in the client is easy, and no dynamic memory allocations are required:

```
Example_var ev = ...;           // Get reference

FLS in;                         // Initialize in param
in.long_val = 99;
in.double_val = 33.0;

FLS inout;                      // Initialize inout param
inout.long_val = 33;
in.double_val = 11.0;

FLS out;                        // Out param
FLS ret_val;                    // Return value

ret_val = op(in, inout, out);   // Make call

// inout may have been changed, and out and ret_val
// contain the values returned by the server.
```

Fixed-Length Array Parameters

Fixed-length array parameters follow the same parameter-passing rules as other fixed-length types. However, an array that is passed in C++ degenerates to a pointer to the first element, so the method signature is expressed in terms of pointers to array slices.

For example, the following IDL defines an operation with fixed-length array parameters:

```
typedef long Larr[3];

interface Example {
  Larr op(in Larr in_p, inout Larr inout_p, out Larr out_p);
};
```

The IDL compiler maps this IDL to the following C++ definitions:

```
typedef CORBA::Long  Larr[3];
typedef CORBA::Long  Larr_slice;
typedef Larr_slice * Larr_out;
// ...
virtual Larr_slice * op(
```

```
    const Larr in_p, Larr_slice * inout_p, Larr_out out_p
) = 0;
```

For `in`, `inout`, and `out` parameters, memory is caller-allocated and need not be on the heap; the method receives and, for `inout` and `out` parameters, modifies the array via the passed pointer. For the return value, a pointer must be returned to dynamically allocated memory, simply because there is no other way to return an array in C++. Therefore, the client must deallocate the return value when it is no longer wanted:

```
Example_var ev = ...;               // Get reference

Larr in = { 1, 2, 3 };              // Initialize in param
Larr inout = { 4, 5, 6 };           // Initialize inout param
Larr out;                           // out param
Larr_slice * ret_val;               // return value
ret_val = ev->op(in, inout, out);   // Make call

// Use results...
Larr_free(ret_val);                 // Must deallocate here!
```

In the previous example, the call to `Larr_free` is required to prevent a memory leak. Alternatively, you can use `_var` types to avoid the need for deallocation. So, you can rewrite the previous example as follows:

```
Example_var ev = ...;               // Get reference

Larr in = { 1, 2, 3 };              // Initialize in param
Larr inout = { 4, 5, 6 };           // Initialize inout param
Larr out;                           // out param, note _var type!
Larr_var ret_val;                   // return value

ret_val = ev->op(in, inout, out);   // Make call

// Use results...

// No need to deallocate anything here, ret_val takes care of it.
```

`_var` types are well-suited to manage the transfer of memory ownership from sender to receiver because they work transparently for both fixed- and variable-length types.

String Parameters

The C++ mapping does not encapsulate strings in a class, so string parameters are passed as `char *`. Because strings are variable-length types, the following memory management issues apply:

- `in` strings are passed as `const char *`, so the callee cannot modify the string's value. The passed string need not be allocated on the heap.
- `inout` strings must be allocated on the heap by the caller. The callee receives a C++ reference to the string pointer. This is necessary because the callee might need to reallocate the string if the new value is longer than the initial value. Passing a reference to the callee lets the callee modify the bytes of the string and the string pointer itself. Responsibility for deallocating the string remains with the caller.
- `out` strings are dynamically allocated by the callee. Responsibility for deallocating the string passes to the caller.
- Strings returned as the return value behave like `out` strings: they are allocated by the callee and responsibility for deallocation passes to the caller.

For example, the following IDL defines an operation with string parameters:

```
interface Example {
    string op(
                in string       in_p,
                inout string    inout_p,
                out string      out_p
            );
};
```

The IDL compiler maps this interface to the following class, in which string parameters are passed as `char *`:

```
class String_out;       // In the CORBA namespace
//...
virtual const char *
op(
    const char *        in_p,
    char * &            inout_p,
    CORBA::String_out   out_p
) = 0;
```

The following example shows how to invoke an operation that passes a string in each possible direction:

```
Example_var ev = ...;                          // Get ref

char * inout = CORBA::string_dup("Hello");    // Initialize
char * out;
char * ret_val;

ret_val = ev->op("Input string", inout, out); // Make call

// Use the strings...

CORBA::string_free(inout);   // We retain ownership
CORBA::string_free(out);     // Caller passed responsibility
CORBA::string_free(ret_val); // Caller passed responsibility
```

This example illustrates the following points:

- The `in` parameter can be allocated anywhere; the example passes a string literal that is allocated in the data segment.
- The caller must pass a dynamically allocated string as the `inout` parameter, because the callee assumes that it can, if necessary, deallocate that parameter.
- The caller must deallocate the `inout` and `out` parameter and the return value.

The following example shows the same method call as before, but uses `String_var` variables to deallocate memory:

```
Example_var ev = ...;

CORBA::String_var inout = CORBA::string_dup("Hello");
CORBA::String_var out;
CORBA::String_var ret_val;

ret_val = ev->op("Input string", inout, out);

// Use the strings...

// No need to deallocate there because the String_var
// variables take ownership.
```

Be careful not to pass a default-constructed `String_var` as an `in` or `inout` parameter:

```
Example_var ev = ...;

CORBA::String_var in;       // Bad: no initialization
CORBA::String_var inout;    // Bad: no initialization
CORBA::String_var out;
CORBA::String_var ret_val;

ret_val = ev->op(in, inout, out);   // Oops :-(
```

In this example, `in` and `inout` are initialized to the null pointer by the default constructor. However, it is illegal to pass a null pointer across an interface; code that does so is liable to crash or raise an exception.

Note: This restriction applies to all types that are passed by pointer, such as arrays and variable-length types. Never pass a null pointer or an uninitialized pointer. Only one exception applies: you can pass a nil reference, even if nil references are implemented as null pointers.

_out Types

IDL `out` parameters result in proxy signatures that use C++ `_out` types. `_out` types ensure correct deallocation of previous results for `_var` types.

For example, the following IDL defines a single out parameter:

```
interface Person {
    void get_name(out string name);
    // ...
};
```

The IDL compiler generates the following class:

```
class Person {
public:
    void get_name(CORBA::String_out name);
    // ...
};
```

The following code fragment uses the `Person` interface, but leaks memory:

```
char * name;
Person_var person_1 = ...;
Person_var person_2 = ...;

person_1->get_name(name);
cout << "Name of person 1: " << name << endl;

person_2->get_name(name);                        // Bad news!
cout << "Name of person 2: " << name << endl;

CORBA::string_free(name);                        // Deallocate
```

Because variable-length `out` parameters are dynamically allocated by the proxy stub, the second call to `get_name()` causes the result of the first `get_name` call to leak.

The following code corrects this problem by deallocating variable-length `out` parameters between invocations:

```
char * name;
Person_var person_1 = ...;
Person_var person_2 = ...;

person_1->get_name(name);
cout << "Name of person 1: " << name << endl;
CORBA::String_free(name);                        // Much better!

person_2->get_name(name);                        // No problem
cout << "Name of person 2: " << name << endl;
CORBA::String_free(name);                        // Deallocate
```

However, if we use `_var` types, no deallocation is required at all:

```
CORBA::String_var name;                       // Note String_var
Person_var person_1 = ...;
Person_var person_2 = ...;

person_1->get_name(name);
cout << "Name of person 1: " << name << endl;

person_2->get_name(name);                     // No leak here
cout << "Name of person 2: " << name << endl;

// No need to deallocate name
```

When the name variable is passed to get_name a second time, the mapping implementation transparently deallocates the previous string. However, how does the mapping manage to avoid deallocation for pointer types but deallocates the previous value for _var types?

The answer lies in the formal parameter type CORBA::String_out, which is a class as outlined here:

```
class String_out {              // In the CORBA namespace
public:
    String_out(char * & s): m_ref(s) { m_ref = 0 }
    String_out(String_var & s): m_ref(s.m_ref) {
        string_free(m_ref);
        m_ref = 0;
    }
    // Other member methods here...
private:
    char * & m_ref;
};
```

This implementation of CORBA::String_out shows how char * out parameters are left alone, but _var out parameters are deallocated.

If you pass a char * as an out parameter, the compiler looks for a way to convert the char * into a String_out object. The single-argument constructor for char * acts as a user-defined conversion operator, so the compiler finds an argument match by constructing a temporary String_out object that is passed to the method. Note that the char * constructor is passed a reference to the string, which it binds to the private member variable m_ref. The constructor body then assigns zero to the m_ref member. m_ref is a reference to the passed string, so construction from a char * clears (sets to null) the actual argument that is passed to the constructor, without deallocating the previous string.

On the other hand, if you pass a String_var as an out parameter, the compiler uses the second constructor to construct the temporary String_out. That constructor binds the m_ref member variable to the passed String_var's internal pointer and deallocates the current string before setting the passed string pointer to null.

_out types are generated for all complex types, such as strings, sequences, and structures. If a complex type has fixed length, then the generated _out type is simply an alias for a reference to the actual type (see "Fixed-Length Complex Parameters" on page 166 for an example).

Note: You can ignore most of the implementation details for _out types. It is only important to know that they serve to prevent memory leaks when you pass a _var as an out parameter.

Variable-Length Complex Parameters

The parameter-passing rules for variable-length complex types differ from those for fixed-length complex types. In particular, for out parameters and return values, the caller is responsible for deallocating the value.

For example, the following IDL defines an operation with variable-length complex parameters:

```
struct VLS {            // Variable-Length Structure
    long    long_val;
    string  string_val;
};

interface Example {
    VLS op(in VLS in_p, inout VLS inout_p, out VLS out_p);
};
```

The IDL compiler maps this IDL to the following C++ definitions:

```
class VLS_out;
// ...
virtual VLS *
op(const VLS & in_p, VLS & inout_p, VLS_out out_p) = 0;
```

The following code calls the op() operation:

```
Example_var ev = ...;           // Get reference

VLS in;                         // Initialize in param
in.long_val = 99;
in.string_val = CORBA::string_dup("Ninety-nine");
```

```
VLS inout;                      // Initialize inout param
inout.long_val = 86;
in.string_val = CORBA::string_dup("Eighty-six");

VLS * out;                      // Note *pointer* to out param
VLS * ret_val;                  // Note *pointer* to return value

ret_val = op(in, inout, out);   // Make call

// Use values...

delete out;                     // Make sure nothing is leaked
delete ret_val;                 // Ditto...
```

As with fixed-length complex types, `in` and `inout` parameters can be ordinary stack variables. However, both the `out` parameter and the return value are dynamically allocated by the call. You are responsible for deallocating these values when you no longer require them.

You can also use `_var` types to take care of the memory-management chores for you, as in this modified version of the previous code:

```
Example_var ev = ...;           // Get reference

VLS in;                         // Initialize in param
in.long_val = 99;
in.string_val = CORBA::string_dup("Ninety-nine");

VLS inout;                      // Initialize inout param
inout.long_val = 86;
in.string_val = CORBA::string_dup("Eighty-six");

VLS_var out;                    // Note _var type
VLS_var ret_val;                // Note _var type

ret_val = op(in, inout, out);   // Make call

// Use values...

// No need to deallocate anything here
```

Note: Type Any is passed using the same rules—that is, out parameters and return values are dynamically allocated by the stub and must be deallocated by the caller. Of course, you can use CORBA::Any_var to achieve automatic deallocation.

Variable-Length Array Parameters

Variable-length arrays are passed as parameters in the same way as fixed-length arrays, except for out parameters: these are passed as a reference to a pointer. As for strings, the generated _out class takes care of deallocating values from a previous invocation held in _var types.

For example, the following IDL defines an operation with variable-length string array parameters:

```
typedef string Sarr[3];

interface Example {
  Sarr op(in Sarr in_p, inout Sarr inout_p, out Sarr out_p);
};
```

The IDL compiler maps this IDL to the following C++ definitions:

```
typedef CORBA::String_mgr Sarr[3];
typedef CORBA::String_Mgr Sarr_slice;
class Sarr_out;
// ...
virtual Sarr_slice * op(
    const Sarr in_p, Sarr_slice * inout_p, Sarr_out out_p
) = 0;
```

The following code calls the op() operation:

```
Example_var ev = ...;       // Get reference

Sarr in;
in[0] = CORBA::string_dup("Bjarne");
in[1] = CORBA::string_dup("Stan");
in[2] = CORBA::string_dup("Andrew");

Sarr inout;
inout[0] = CORBA::string_dup("Dennis");
```

```
inout[1] = CORBA::string_dup("Ken");
inout[2] = CORBA::string_dup("Brian");

Sarr_slice * out;                   // Pointer to array slice
Sarr_slice * ret_val;               // Pointer to array slice

ret_val = ev->op(in, inout, out);   // Make call

// Use values...

Sarr_free(out);                     // Deallocate to avoid leak
Sarr_free(ret_val);                 // Ditto...
```

As always, you can rewrite the code to use _var types, and so prevent memory leaks:

```
Example_var ev = ...;       // Get reference

Sarr in;
in[0] = CORBA::string_dup("Bjarne");
in[1] = CORBA::string_dup("Stan");
in[2] = CORBA::string_dup("Andrew");

Sarr inout;
inout[0] = CORBA::string_dup("Dennis");
inout[1] = CORBA::string_dup("Ken");
inout[2] = CORBA::string_dup("Brian");

Sarr_var out;                       // Note _var type
Sarr_var ret_val;                   // Note _var type

ret_val = ev->op(in, inout, out);   // Make call

// Use values...

// No need to free anything here
```

Object Reference Parameters

You pass object references as parameters as you do strings. For `inout` reference, the caller must pass a C++ reference to a `_ptr` reference. For an out parameters and return values, the caller is responsible for deallocation.

For example, the following IDL defines an operation with object reference parameters:

```
interface Example {
    string greeting();
    Example op(
        in Example      in_p,
        inout Example   inout_p,
        out Example     out_p
    );
};
```

The IDL compiler maps this IDL to the following C++ definitions:

```
class Example_out;
// ...
virtual Example_ptr op(
  Example_ptr in_p, Example_ptr & inout_p, Example_out out_p
) = 0;
```

The following code calls the `op()` operation:

```
Example_var ev = ...;
Example_var in = ...;       // Initialize in param
Example_var inout = ...;    // Initialize inout param
Example_ptr out;            // Note _ptr reference
Example_ptr ret_val;        // Note _ptr reference

ret_val = ev->op(in, inout, out);

// Use references...

CORBA::release(out);        // Deallocate
CORBA::release(ret_val);    // Ditto...
```

Note that the code explicitly releases the references returned as the `out` parameter and the return value.

You can also rewrite this code to use `_var` references in order to avoid memory leaks:

```
Example_var ev = ...;
Example_var in = ...;       // Initialize in param
Example_var inout = ...;    // Initialize inout param
Example_var out;            // Note _var reference
Example_var ret_val;        // Note _var reference
```

```
ret_val = ev->op(in, inout, out);

// Use references...

// No need to deallocate here
```

Parameter-Passing Rules: Summary

The following sections summarize the parameter-passing rules for the C++ mapping.

Never Pass Null or Uninitialized Pointers as in or inout Parameters.

As shown earlier (see page 171), it is illegal to pass null pointers or uninitialized pointers as `inout` or `in` parameters. The most likely outcome of ignoring this rule is a core dump.

Nil object references are exempt from this rule, so it is safe to pass a nil reference as a parameter.

Do Not Ignore Variable-Length Return Values

Ignoring return values can leak memory. For example, the following interface defines operation `do_something()` to return a string value:

```
// interface Example {
//    string do_something();
// };
```

The following client call on `do_something()` erroneously ignores its return value:

```
Example_var ev = ...;      // Get reference
ev->do_something();        // Memory leak!
```

Be careful never to ignore the return, because the memory that the stub allocates to the return value can never be reclaimed.

Allocate String and Reference inout Parameters on the Heap and Deallocate them After the Call

String and reference `inout` parameters must be allocated on the heap; ownership of the memory remains with the caller.

Deallocate Variable-Length Return Values and out Parameters

Variable-length types passed as return values or `out` parameters are passed by pointer and are dynamically allocated by the stub. You must deallocate these values to avoid memory leaks.

Use _var Types for Complex inout and out Parameters and Return Values

Always use a `_var` type when a value must be heap-allocated. This includes any complex or variable-length `inout` or `out` parameter or return value. After you have assigned a parameter to a `_var` type, you don't have to worry about deallocating memory.

For example, the following interface defines three operations:

```
// Some sample IDL to show how _var types make life easier.
interface Example {
    string  get_string();
    void    modify_string(inout string s);
    void    put_string(in string s);
};
```

Because `_var` types convert correctly to pass in any direction, the following code does exactly the right things:

```
// _var automates memory management.
{
    Example_var ev = ...;   // Get reference
    CORBA::String_var s;    // Parameter

    s = ev->get_string();   // Get value
    ev->modify_string(s);   // Change it
    ev->put_string(s);      // Put it somewhere
}
// Everything is deallocated here
```

Table 10 summarizes parameter-passing rules. It does not show that out parameters are passed as _out types. Instead, it shows the corresponding alias for fixed-length types, or the type of constructor argument for the _out type for variable-length types.

Table 10: *Parameter passing for low-level mapping*

IDL Type	in	inout	out	Return Value
simple	*simple*	*simple* &	*simple* &	*simple*
enum	*enum*	*enum* &	*enum* &	*enum*
fixed	const Fixed &	Fixed &	Fixed &	Fixed
string	const char *	char * &	char * &	char *
wstring	const WChar *	WChar * &	WChar * &	WChar *
any	const Any &	Any &	Any * &	Any *
objref	*objref_ptr*	*objref*_ptr &	*objref*_ptr &	*objref*_ptr
sequence	const *sequence* &	*sequence* &	*sequence* * &	*sequence* *
struct, fixed	const *struct* &	*struct* &	*struct* &	*struct*
union, fixed	const *union* &	*union* &	*union* &	*union*
array, fixed	const *array*	*array*_slice *	*array*_slice *	*array*_slice *
struct, variable	const *struct* &	*struct* &	*struct* * &	*struct* *
union, variable	const *union* &	*union* &	*union* * &	*union* *
array, variable	const *array*	*array*_slice *	*array*_slice * &	*array*_slice *

As Table 10 shows, the parameter type varies for both out parameters and return values, depending on whether a complex structure, union, or array is variable length or fixed length. Table 11 shows the considerably simpler parameter-passing rules for _var types:

Table 11: *Parameter passing with _var types*

IDL Type	in	inout/out	Return Value
string	const String_var &	String_var &	String_var
wstring	const WString_var &	WString_var &	WString_var
any	const Any_var &	Any_var &	Any_var
objref	const *objref*_var &	*objref*_var &	*objref*_var
sequence	const *sequence*_var &	s*sequence*_var &	*sequence*_var
struct	const *struct*_var &	*struct*_var &	*struct*_var
union	const *union*_var &	*union*_var &	*union*_var
array	const *array*_var &	*array*_var &	*array*_var

_var types are carefully crafted so that parameter passing is uniform, regardless of the underlying type. This aspect of _var types, together with their automatic deallocation behavior, makes them most useful for parameter passing.

Setting 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:

- On the client ORB, so they apply to all invocations.
- On a given thread, so they apply only to invocations on that thread
- On individual objects, so they apply only to invocations on those objects.

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 138.

Note: Because all policy types and their settings are defined in the Messaging module, client code that sets quality of service policies must include `omg/messaging.hh`.

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 12 shows which policies are supported in each category:

Table 12: *Timeout Policies*

OMG Timeout Policies	RelativeRoundtripTimeoutPolicy ReplyEndTimePolicy RelativeRequestTimeoutPolicy RequestEndTimePolicy
Proprietary Timeout Policies	BindingEstablishmentPolicy RelativeBindingExclusiveRoundtripTimeoutPolicy RelativeBindingExclusiveRequestTimeoutPolicy 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 Times

Two policies, RequestEndTimePolicy and ReplyEndTimePolicy, set absolute deadlines for request and reply delivery, respectively, through the TimeBase::UtcT type. The Orbix libraries include helper class IT_UtcT, which provides ease-of-use operators and methods for working with the types defined in the

TimeBase module. For example, you can use IT_UtcT::current() and IT_UtCT::operator+() to obtain an absolute time that is relative to the current time.

For more information, refer to the *Orbix 2000 Programmer's Reference*.

RelativeRoundtripTimeoutPolicy

This policy 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:

```
// C++
TimeBase::TimeT relative_expiry = 4L * 10000000L; // 4 seconds
try{
    CORBA::Any relative_roundtrip_timeout_value;
    relative_roundtrip_timeout_value <<= relative_expiry;
    CORBA::PolicyList policies(1);
    policies.length(1);
    policies[0] = orb->create_policy(
        Messaging::RELATIVE_RT_TIMEOUT_POLICY_TYPE,
        relative_roundtrip_timeout_value
    );
    policy_manager->set_policy_overrides(
        policies,
        CORBA::ADD_OVERRIDE
    );
}
catch (CORBA::PolicyError& pe){
    return 1;
```

```
}
catch (CORBA::InvalidPolicies& ip){
    return 1;
}
catch (CORBA::SystemException& se){
    return 1;
}
```

ReplyEndTimePolicy

This policy 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 Times" on page 185).

No default is set for this policy; if it is not set, a request has unlimited time to complete.

RelativeRequestTimeoutPolicy

This policy 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:

```
// C++
TimeBase::TimeT relative_expiry = 3L * 10000000L; // 3 seconds
try{
    CORBA::Any relative_request_timeout_value;
    relative_request_timeout_value <<= relative_expiry;
    CORBA::PolicyList policies(1);
    policies.length(1);
```

```
    policies[0] = orb->create_policy(
        Messaging::RELATIVE_REQ_TIMEOUT_POLICY_TYPE,
        relative_request_timeout_value
    );
    policy_manager->set_policy_overrides(
        policies,
        CORBA::ADD_OVERRIDE
    );
}
catch (CORBA::PolicyError& pe){
    return 1;
}
catch (CORBA::InvalidPolicies& ip){
    return 1;
}
catch (CORBA::SystemException& se){
    return 1;
}
```

RequestEndTimePolicy

This policy 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 Times" on page 185).

No default is set for this policy; if it is not set, request delivery has unlimited time to complete.

BindingEstablishmentPolicy

This policy 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:

```
// C++
try{
    CORBA::Any bind_est_value;

    IT_CORBA::BindingEstablishmentPolicyValue val;
    val.rel_expiry    = (TimeBase::TimeT)30 * 10000000; // 30s
    val.max_rebinds   = (CORBA::UShort)5;    // 5 binding tries
    val.max_forwards  = (CORBA::UShort)20;   // 20 forwards
    val.initial_iteration_delay
                      = (TimeBase::TimeT)1000000; // 0.1s delay
    val.backoff_ratio = (CORBA::Float)2.0;       // back-off ratio

    bind_est_value <<= val;

    CORBA::PolicyList policies(1);
    policies.length(1);
    policies[0] = orb->create_policy(
        IT_CORBA::BINDING_ESTABLISHMENT_POLICY_ID,
        bind_est_value
    );

    CORBA::Object_var obj = slave->_set_policy_overrides(
        policies,
        CORBA::ADD_OVERRIDE
    );

    lots_of_retries_slave = ClientPolicy::Slave::_narrow(obj);
}
catch (CORBA::PolicyError& pe){
    return 1;
}
catch (CORBA::InvalidPolicies& ip){
    return 1;
}
catch (CORBA::SystemException& se){
    return 1;
}
```

RelativeBindingExclusiveRoundtripTimeoutPolicy

This policy 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

This policy 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.

InvocationRetryPolicy

This policy 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:

```
// C++
try{
    CORBA::Any lots_of_retries_value;

    IT_CORBA::InvocationRetryPolicyValue val;
    val.max_retries   = (CORBA::UShort)10000;   // 10000 retries
    val.max_rebinds   = (CORBA::UShort)5;       // 5 rebinds
    val.max_forwards  = (CORBA::UShort)20;      // 20 forwards
```

```
    val.initial_retry_delay
                    = (TimeBase::TimeT)1000000; // 0.1s delay
    val.backoff_ratio = (CORBA::Float)2.0;       // back-off ratio

    lots_of_retries_value <<= val;

    CORBA::PolicyList policies(1);
    policies.length(1);
    policies[0] = orb->create_policy(
        IT_CORBA::INVOCATION_RETRY_POLICY_ID,
        lots_of_retries_value
    );

    CORBA::Object_var obj = slave->_set_policy_overrides(
        policies,
        CORBA::ADD_OVERRIDE
    );

    lots_of_retries_slave = ClientPolicy::Slave::_narrow(obj);
}
catch (CORBA::PolicyError& pe){
    return 1;
}
catch (CORBA::InvalidPolicies& ip){
    return 1;
}
catch (CORBA::SystemException& se){
    return 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 202). 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`.

9 Developing a Server

This chapter explains how to develop a server that implements servants for CORBA objects.

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.
- Manages memory for POA servants and object references.
- Initializes and shuts down the runtime ORB.
- Passes parameters to server-side operations.

For an overview of server code requirements, see "Learning More About the Server" on page 61. Although throwing exceptions is an important aspect of server programming, it is covered separately in Chapter 13.

For information on ORB initialization and shutdown, see "ORB Intialization and Shutdown" on page 129.

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.

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 10.

For each IDL interface, the IDL compiler generates a POA_ skeleton class that you compile into the server application. Skeleton classes are abstract base classes. You implement skeleton classes in the server application code with servant classes, which define the behavior of the pure virtual 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.

The IDL compiler can also generate a TIE class, which lets you implement CORBA objects with classes that are unrelated (by inheritance) to skeleton classes. For more information, see "Delegating Servant Implementations" on page 214.

Note: The POA_ prefix only applies to the outermost naming scope of an IDL construct. So, if an interface is nested in a module, only the outermost module gets the POA_ prefix; constructs nested inside the module do not have the prefix.

Figure 20 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.

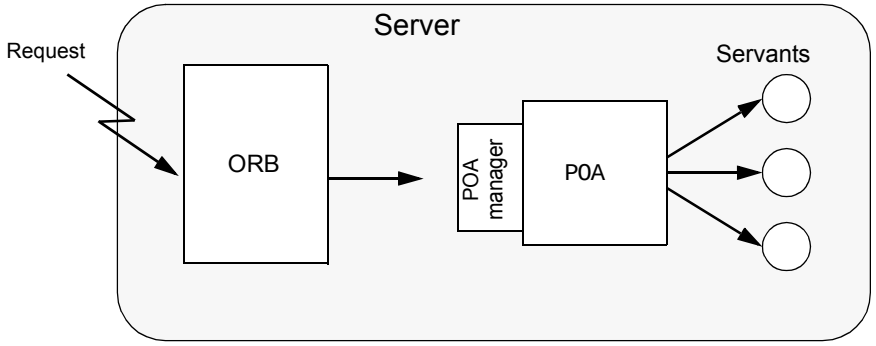

Figure 20: *The server-side ORB conveys client requests to the POA via its manager, and the POA dispatches the request to the appropriate servant.*

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. All operations are represented as virtual member methods, so dynamic binding ensures that the proper method in your derived servant class is invoked.

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 skeleton class `POA_BankDemo::Account`. For purposes of simplification, only methods that map directly to IDL operations and attribute are shown:

```
namespace POA_BankDemo
{
    class Account :
        virtual public PortableServer::ServantBase

    {
        virtual ::BankDemo::AccountId
        account_id() IT_THROW_DECL((CORBA::SystemException)) = 0;

        virtual ::BankDemo::CashAmount
        balance() IT_THROW_DECL((CORBA::SystemException)) = 0;

        virtual void
        withdraw(
            ::BankDemo::CashAmount amount
        ) IT_THROW_DECL((CORBA::SystemException,
          BankDemo::Account::InsufficientFunds)) = 0;

        virtual void
        deposit(
            ::BankDemo::CashAmount amount
        ) IT_THROW_DECL((CORBA::SystemException)) = 0;
    };
```

The following points are worth noting about the skeleton class:

- `POA_BankDemo::Account` inherits from `PortableServer::ServantBase`. All skeleton classes inherit from the `ServantBase` class for two reasons:
  - `ServantBase` 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 `ServantBase`.

- The names of the skeleton class and the corresponding client-side proxy class are different. In this case, the fully scoped name of the skeleton class is `POA_BankDemo::Account`, while the proxy class name is `BankDemo::Account`.

  This differentiation is important if client and server are linked into the same program, because it avoids name clashes for multiply defined symbols. It also preserves location transparency because it guarantees that collocated calls are always dispatched by an intervening proxy object, and are never dispatched as a direct virtual method call from client to servant. So, if the server decides to delete an object and a collocated client attempts to make a call on the deleted object, the proxy raises an `OBJECT_NOT_EXIST` exception instead of attempting to access deallocated memory and causing the program to crash.
- The skeleton class defines methods that correspond to the interface operations and attributes.
- Methods are all defined as pure virtual, so you cannot instantiate a skeleton class. Instead, you must derive from the skeleton a concrete servant class that implements the pure virtual methods that it inherits.
- Each method has an exception specification. Orbix generates exception specifications only for skeleton classes. In this example, the methods throw system exceptions and, in the case of `withdraw()`, the user exception `InsufficientFunds`.
- The `throw` clause prevents methods from throwing illegal exceptions. For example, if `deposit()` throws an exception other than `CORBA::SystemException`, the C++ run time calls the `unexpected` method (which, by default, aborts the process).
- Apart from the exception specification, the signature of each skeleton class method is the same as the corresponding proxy class method.

  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 derives from skeleton class `POA_BankDemo::Account`. Unlike the skeleton class methods, the `AccountImpl` methods that map to IDL operations and attributes are not pure virtual, so a server can instantiate `AccountImpl` as a servant.

```
#include "BankDemoS.hh"  // Generated server-side header

class AccountImpl : public POA_BankDemo::Account {
public:
    // Inherited IDL operations

    virtual BankDemo::AccountId
    account_id() IT_THROW_DECL((CORBA::SystemException));

    virtual BankDemo::CashAmount
    balance() IT_THROW_DECL((CORBA::SystemException));

    virtual void
    withdraw(
        BankDemo::CashAmount amount
    ) IT_THROW_DECL((CORBA::SystemException,
      BankDemo::Account::InsufficientFunds));

    virtual void
    deposit(
        BankDemo::CashAmount amount
    ) IT_THROW_DECL((CORBA::SystemException));

    // other members here ...

private:
    // Prevent copying and assigment of servants
    AccountImpl(const AccountImpl &);
    void operator=(const AccountImpl &);
};
```

The following requirements and recommendations apply to servant class definitions:

- The code must include the generated server header file—in this case, `BankDemoS.hh`.
- `AccountImpl` inherits from `POA_BankDemo::Account` through virtual inheritance. If, as in this case, the servant class inherits from only one source, it is unimporant to specify virtual inheritance. However, a servant class that inherits from multiple skeleton classes should always use virtual inheritance to prevent errors.
- 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.
- The copy constructor and assignment operator for the servant class are private to prevent copying and assignment of servant instances.

  Servants should not be copied or assigned; only one servant should incarnate any given CORBA object; otherwise, it is unclear which servant should handle requests for that object. It is always good practice to hide a servant's copy constructor and assignment operator.

The preceding `AccountImpl` class is a complete and functional servant class. It only remains to implement the pure virtual methods that are inherited from the skeleton. You can also can add other member variables and methods, public and private, that can help implement a servant. For example, it is typical to add a constructor and destructor, and private member variables to hold the state of the object while the servant is in memory.

Implementing Operations

Most work in developing a servant consists of implementing each inherited pure virtual method. Because the application code controls the body of each operation, it largely determines the application's overall behavior. The following code outlines an implementation of the `withdraw()` method:

```
void
AccountImpl::withdraw(
    BankDemo::CashAmount amount
) IT_THROW_DECL((
    CORBA::SystemException,
    BankDemo::Account::InsufficientFunds
))
```

```
{
    // ... database connection (via PSS) code omitted here

    // get a PSS reference to corresponding database object
    IT_PSS_RefVar<BankDemoStore_AccountBaseRef> ref =
        my_state(accounts_home_obj.in());

    BankDemo::CashAmount new_balance = ref->balance() - amount;

    if (new_balance < 0.0F)
    {
        cout << " throwing InsufficientFunds" <<  endl;
        throw BankDemo::Account::InsufficientFunds();
    }

    ref->balance(new_balance);
    // ...

    cout << " withdrew $" << amount << endl;
}
```

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. The easiest way to do this is to call `_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.

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 238).

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 "Explicit and Implicit Object Activation" on page 236.

Note: The object reference returned by `_this()` is independent of the servant itself; you must eventually call `release()` on the object or hold it in a `_var` reference in order to avoid resource leaks. Releasing the object reference has no effect on the corresponding servant.

Handling Output Parameters

Server-side rules for passing output (`in`/`inout`) parameters and return values to the client complement client-side rules. For example, if the client is expected to deallocate a variable-length return value, the server must allocate that value.

In general, these rules apply:

- If the type to pass is variable-length, the server dynamically allocates the value and the client deallocates it.
- String, reference, and variable-length array types are dynamically allocated and deallocated by the client. Strings and references can be reallocated by the server.

Other types are passed by value or reference.

The following sections show the server-side rules for passing output parameters and return values of various IDL types.

Simple Parameters

Simple IDL types such as `short` or `long` are passed by value. For example, the following IDL defines operation `Example::op()`, which passes three `long` parameters:

```
interface Example {
    long
    op( in long in_p, inout long inout_p, out long out_p);
};
```

The corresponding servant class contains this signature for `op()`:

```
virtual CORBA::Long
op(
    CORBA::Long     in_p,
    CORBA::Long &   inout_p,
    CORBA::Long_out out_p
) throw(CORBA::SystemException);
```

This example has the same mapping as the client, where `CORBA::Long_out` type is simply an alias for `CORBA::Long &`. You might implement this operation as follows:

```
CORBA::Long
ExampleImpl::op(
   CORBA::Long in_p, CORBA::Long & inout_p, CORBA::Long_out out_p
) throw(CORBA::SystemException)
{
    inout_p = 2 * inout_p;       // Change inout_p.
    out_p = in_p * in_p;         // Set out_p
    return in_p / 2;             // Return in_p
}
```

The method simply sets output parameters and return values; the changes are automatically propagated back to the client.

Fixed-Length Complex Parameters

Fixed-length complex parameters are passed by value or by reference. For example, the following IDL defines a fixed-length structure that operation `Example::op()` uses in its return value and parameters:

```
struct FLS {            // Fixed-Length Structure
    long    long_val;
    double  double_val;
};

interface Example {
    FLS op(in FLS in_p, inout FLS inout_p, out FLS out_p);
};
```

The corresponding servant class contains this signature for `op()`:

```
typedef FLS & FLS_out;
// ...
virtual FLS
op(const FLS & in_p, FLS & inout_p, FLS_out out_p)
throw(CORBA::SystemException);
```

The following code implements the servant operation. No memory management issues arise; the method simply assigns the values of output parameters and the return value:

```
FLS
ExampleImpl::op(const FLS & in_p, FLS & inout_p, FLS_out out_p)
throw(CORBA::SystemException)
{
    cout << in_p.long_val << endl;      // Use in_p
    cout << in_p.double_val << endl;    // Use in_p
    cout << inout_p.double_val << endl; // Use inout_p

    // Change inout_p
    inout_p.double_val = inout_p.long_val * in_p.double_val;

    out_p.long_val = 99;        // Initialize out_p
    out_p.double_val = 3.14;
```

```
    FLS ret_val = { 42, 42.0 }; // Initialize return value
    return ret_val;
}
```

Fixed-Length Array Parameters

Fixed-length arrays are passed as pointers to array slices. The return value is dynamically allocated. For example, the following IDL defines a fixed-length array that operation `Example::op()` uses in its return value and parameters:

```
typedef long Larr[3];

interface Example {
  Larr op(in Larr in_p, inout Larr inout_p, out Larr out_p);
};
```

The corresponding servant class contains this signature for `op()`:

```
typedef CORBA::Long  Larr[3];
typedef CORBA::Long  Larr_slice;
typedef Larr_slice * Larr_out;
// ...
virtual Larr_slice *
op(const Larr in_p, Larr_slice * inout_p, Larr_out out_p)
throw(CORBA::SystemException);
```

In the following implementation, the generated `Larr_alloc()` method dynamically allocates the return value:

```
Larr_slice *
ExampleImpl::
op(const Larr in_p, Larr_slice * inout_p, Larr_out out_p)
throw(CORBA::SystemException)
{
    int len = sizeof(in_p) / sizeof(*in_p);

    // Use incoming values of in_p and inout_p...

    // Modify inout_p
    inout_p[1] = 12345;

    // Initialize out_p
    for (int i = 0; i < len; i++)
        out_p[i] = i * i;
```

```
    // Return value must be dynamically allocated
    Larr_slice * ret_val = new Larr_alloc();
    for (int i = 0; i < len; i++)
        ret_val[i] = i * i * i;

    return ret_val;
}
```

String Parameters

String-type output parameters and return values must be dynamically allocated. For example, the following IDL defines a fixed-length array that operation `Example::op()` uses in its return value and parameters:

```
interface Example {
    string op(
                in string       in_p,
                inout string    inout_p,
                out string      out_p
            );
};
```

The corresponding servant class contains this signature for `op()`:

```
virtual const char *
op(
    const char *        in_p,
    char * &            inout_p,
    CORBA::String_out   out_p
) throw(CORBA::SystemException);
```

The server is constrained by the same memory requirements as the client:

- Strings are initialized as usual.
- `inout` strings are dynamically allocated and initialized by the client. The servant can change an `inout` string by modifying the bytes of the `inout` string in place, or shorten the `inout` string in place by writing a terminating NUL byte into the string. To return an `inout` string that is longer than the initial value, the servant must deallocate the original copy and allocate a longer string.
- `out` strings must be dynamically allocated.
- Return value strings must be dynamically allocated.

The following code implements the servant operation:

```
const char *
ExampleImpl::
op(
    const char *        in_p,
    char * &            inout_p,
    CORBA::String_out   out_p
) throw(CORBA::SystemException)
{
    cout << in_p << endl;       // Show in_p
    cout << inout_p << endl;    // Show inout_p

    // Modify inout_p in place:
    //
    char * p = inout_p;
    while (*p != '\0')
        toupper(*p++);

    // OR make a string shorter by writing a terminating NUL:
    //
    *inout_p = '\0';            // Set to empty string.

    // OR deallocate the initial string and allocate a new one:
    //

    CORBA::string_free(inout_p);
    inout_p = CORBA::string_dup("New string value");

    // out strings must be dynamically allocated.
    //
    out_p = CORBA::string_dup("I am an out parameter");

    // Return value strings must be dynamically allocated.
    //
    char * ret_val
        = CORBA::string_dup("In Xanadu did Kubla Khan..."));

    return ret_val;
}
```

Variable-Length Complex Parameters

out parameters and return values of variable-length complex types must be dynamically allocated; in and inout parameters are passed by reference.

For example, the following IDL defines a variable-length structure that operation Example::op() uses in its return value and parameters:

```
struct VLS {            // Variable-length structure
    long    long_val;
    string  string_val;
};

interface Example {
    VLS op(in VLS in_p, inout VLS inout_p, out VLS out_p);
};
```

The corresponding servant class contains this signature for op():

```
class VLS_out { /* ... */ };
// ...
virtual VLS *
op(const VLS & in_p, VLS & inout_p, VLS_out out_p)
throw(CORBA::SystemException);
```

The following code implements the servant operation:

```
VLS *
ExampleImpl::
op(const VLS & in_p, VLS & inout_p, VLS_out out_p)
throw(CORBA::SystemException)
{
    cout << in_p.string_val << endl;    // Use in_p
    cout << inout_p.long_val << endl;   // Use inout_p
    inout_p.long_val = 99;              // Modify inout_p
    out_p = new VLS;                    // Allocate out param
    out_p->long_val = 1;                // Initialize...
    out_p->string_val = CORBA::string_dup("One");

    VLS * ret_val = new VLS;            // Allocate return value
    ret_val->long_val = 2;             // Initialize...
    ret_val->string_val = CORBA::string_dup("Two");

    return ret_val;
}
```

Variable-Length Array Parameters

Like fixed-length arrays, variable-length arrays are passed as pointers to array slices. `out` parameters and the return value must be dynamically allocated.

For example, the following IDL defines a variable-length array that operation `Example::op()` uses in its return value and parameters:

```
typedef string Sarr[3];

interface Example {
  Sarr op(in Sarr in_p, inout Sarr inout_p, out Sarr out_p);
};
```

The corresponding servant class contains this signature for `op()`:

```
typedef CORBA::String_mgr Sarr[3];
typedef CORBA::String_Mgr Sarr_slice;
class Sarr_out { /* ... */ };
// ...
virtual Sarr_slice * op(
    const Sarr in_p, Sarr_slice * inout_p, Sarr_out out_p
) throw(CORBA::SystemException);
```

The following code implements the servant operation. As with all nested strings, string elements behave like a `String_var`, so assignments make deep copies or, if a pointer is assigned, take ownership:

```
typedef CORBA::String_mgr Sarr[3];
typedef CORBA::String_Mgr Sarr_slice;
class Sarr_out;
// ...

Sarr_slice *
ExampleImpl::
op(
    const Sarr in_p, Sarr_slice * inout_p, Sarr_out out_p
) throw(CORBA::SystemException)
{
    cout << in_p[1] << endl;    // Use in_p
    cout << inout_p[0] << endl; // Use inout_p
    inout_p[1] = in_p[0];       // Modify inout_p

    out_p = Sarr_alloc();       // Allocate out param
    out_p[0] = CORBA::string_dup("In Xanadu did Kubla Khan");
```

```
    out_p[1] = CORBA::string_dup("A stately pleasure-dome
    out_p[2] = CORBA::string_dup("decree: Where Alph...");

    // Allocate return value and initialize...
    //
    Sarr_slice * ret_val = Sarr_alloc();
    ret_val[0] = out_p[0];
    ret_val[1] = inout_p[1];
    ret_val[2] = in_p[2];

    return ret_val;             // Poor Coleridge...
}
```

Object Reference Parameters

Object references are passed as `_ptr` references. The following memory management rules apply to object reference parameters:

- `in` parameters are initialized by the caller and must not be released; the caller retains ownership of the `in` parameter.
- `inout` parameters are initialized by the caller. To change the value of an `inout` parameter, you must call `release()` on the original value and use `_duplicate()` to obtain the new value.
- `out` parameters and return values must be allocated by `_duplicate()` or `_this()`, which calls `_duplicate()` implicitly.

For example, the following IDL defines interface `Example`; operation `Example::op()` specifies this interface for its return value and parameters:

```
interface Example {
    string greeting();
    Example op(
        in Example      in_p,
        inout Example   inout_p,
        out Example     out_p
    );
};
```

The corresponding servant class contains this signature for `op()`:

```
class Example_out { /* ... */ };
// ...
virtual Example_ptr op(
  Example_ptr in_p, Example_ptr & inout_p, Example_out out_p
) throw(CORBA::SystemException);
```

The following implementation dynamically allocates the new value of `inout_p` after releasing the previous value. The return value is dynamically allocated because `_this()` calls `_duplicate()` implicitly.

As shown in this example, you should always test for nil before making a call on a passed `in` or `inout` reference. Otherwise, your servant is liable to make a call on a nil reference and cause a core dump.

```
Example_ptr
ExampleImpl::
op(
    Example_ptr in_p, Example_ptr & inout_p, Example_out out_p
) throw(CORBA::SystemException)
{
    // Use in_p.
    //
    if (!CORBA::is_nil(in_p)) {
        CORBA::String_var s = in_p->greeting();
        cout << s << endl;
    }

    // Use inout_p.
    //
    if (!CORBA::is_nil(inout_p)) {
        CORBA::String_var s = inout_p->greeting();
        cout << s << endl;
    }

    // Modify inout_p to be the same as in_p.
    //
    CORBA::release(inout_p);                // First deallocate,
    inout_p = Example::_duplicate(in_p);    // then assign.

    // Set return value.
    //
    return _this();         // Return reference to self.
}
```

Note: This example is unrealistic in returning a reference to self, because in order to invoke the operation, the caller must hold a reference to this object already.

Counting Servant References

Multi-threaded servers need to reference-count their servants in order to avoid destroying a servant on one thread that is still in use on another. In general, you should enable reference counting for servants that are activated in a POA with a policy of `ORB_CTRL_MODEL`.

The POA specification provides the standard methods `_add_ref()` and `_remove_ref()` to support reference counting, but by default they do nothing. You can enable reference counting by inheriting the standard class `PortableServer::RefCountServantBase` in servant implementations. For example:

```
class BankDemo_AccountImpl
     : public virtual POA_BankDemo::Account,
       public virtual PortableServer::RefCountServantBase
```

With reference counting enabled, the POA calls `_add_ref()` when it holds a pointer to a servant in any thread, and calls `_remove_ref()` when it is finished with that servant. POA methods that return servants to user code call `_add_ref()` before they deliver the servant, so the same code should call `_remove_ref()` on the result when it is finished.

In your own code, you should call `_add_ref()` for each additional pointer to a servant, and `_remove_ref()` when you are done with that pointer (rather than delete it). Doing so ensures that the servant is deleted when no pointers are held to that servant either in your own code or in the POA.

Reference counting is ignored by tie-based servants. Tie templates, as defined in the POA standard, do not support reference counting, Therefore, it is not recommended that you use the tie approach for multi-threaded servers.

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 template classes. The tie approach lets you implement CORBA objects with classes that are unrelated (by inheritance) to skeleton classes.

In most cases, inheritance and tie approaches are functionally equivalent; only programming style preferences determine whether to favor one approach over the other. For more on the comparative merits of each approach, see "Tie Versus Inheritance" on page 215.

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 template 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. Pass the tie object's address to the tie object constructor with this syntax:

   ```
   tie-template-class<impl-class> tie-servant(tied-object);
   ```

For example, given an IDL specification that includes interface `BankDemo::Bank`, the IDL compiler can generate tie template class `POA_BankDemo::Bank_tie`. 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
POA_BankDemo::Bank_tie<BankImpl> bank_srv_tie(new BankImpl);
```

Given this tie servant, you can use it to create an object reference:

```
//create an object reference for bank servant
bank_var 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 `PortableServer::POA::deactivate_object()`. If the tie servant's tie object implements only a single object, the tie object is also removed.

Tie Versus Inheritance

The tie approach can be useful where implementations must inherit from an existing class framework, as often occurs with OODB systems. In this case, you can create object implementations only with the tie approach. Otherwise, the tie approach has several drawbacks:

- Because the tie approach requires two C++ instances for each CORBA object, it uses up more resources.
- Tie-based servants ignore reference counting; therefore, you should not use the tie approach for multi-threaded servers.
- The tie approach adds an unnecessary layer of complexity to application code.

In general, unless you have a compelling reason to use the tie approach, you should favor the inheritance approach in your code.

Implementation Inheritance

IDL inheritance does not constrain your options for implementing servant classes. In Figure 21, shaded classes represent the skeleton abstract base classes generated by the IDL compiler; non-shaded classes represent the servant classes that you provide

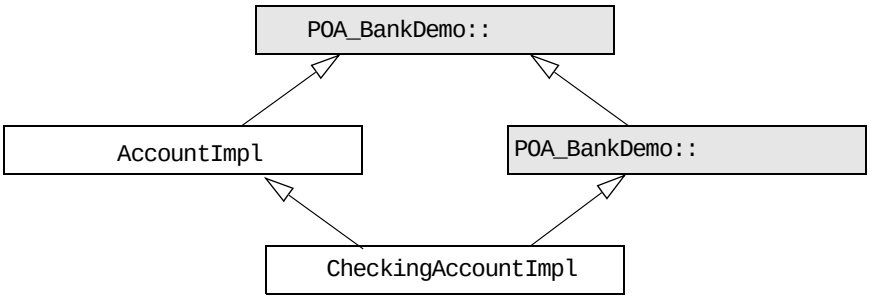

Figure 21: *A servant class can inherit base class implementations.*

`CheckingAccountImpl` inherits from `AccountImpl`, so `CheckingAccountImpl` needs only to implement the two pure virtual methods that it inherits from `CheckingAccount`: `overdraftLimit()` and `orderCheckBook()`. Functions in base interface `Account` such as `balance()` are already implemented in and inherited from `AccountImpl`.

Interface Inheritance

You can choose not to derive `CheckingAccountImpl()` from `AccountImpl()`. If all methods in `POA_BankDemo::CheckingAccount` are defined as pure virtual, then `CheckingAccountImpl` must implement the methods that it inherits from `POA_BankDemo::Account`, as well as those inherited from `POA_BankDemo::CheckingAccount`, as shown in Figure 22

Interface inheritance facilitates encapsulation. With interface inheritance, the derived class servant is independent of the base class servant. This might be desirable if you plan to split a single server into two servers: one that implements base objects and another that implements derived objects.

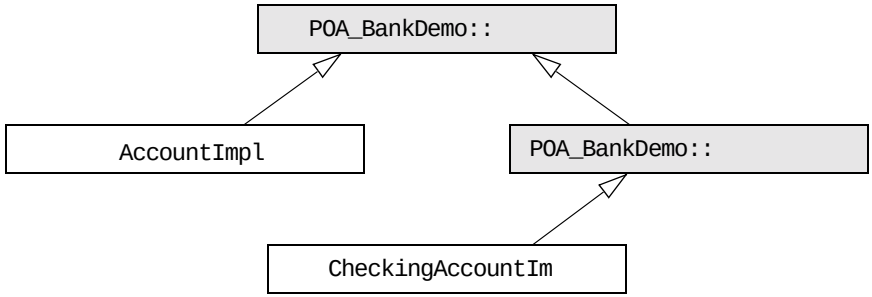

Figure 22: *A servant class can implement operations of all base skeleton classes.*

This model also serves any application design that requires all base classes to be abstract, while it retains interface inheritance.

Multiple Inheritance

Implementation and interface inheritance extend to multiple inheritance. In Figure 23, solid arrows indicate inheritance that is mandated by the C++ mapping. The dotted arrows indicate that the servants allow either implementation or interface inheritance.

Given this hierarchy, it is also possible to leave `POA_BankDemo::Account` without an implementation, inasmuch as it is an IDL abstract base class. In this case, `CheckingAccountImpl` and `SavingsAccountImpl` must provide the required virtual method implementations.

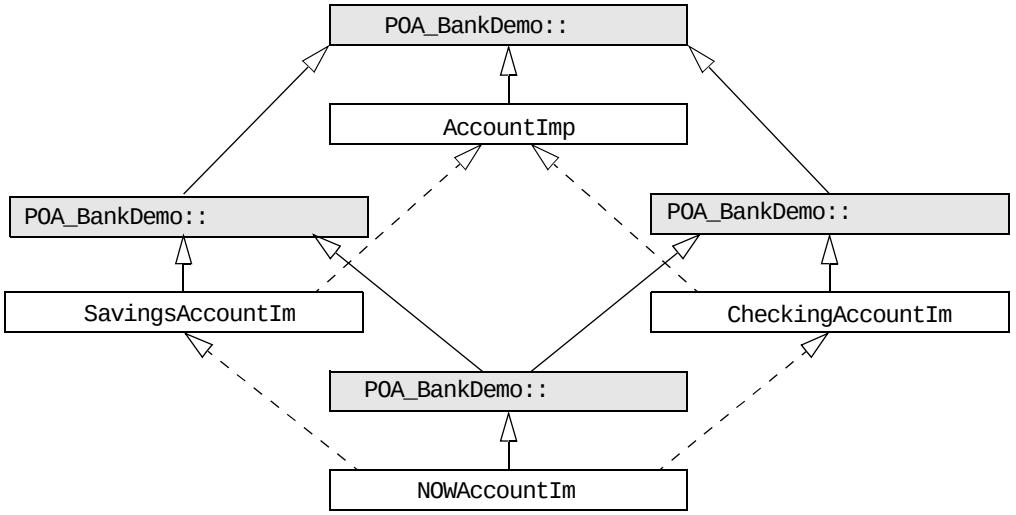

Figure 23: *Inheritance options among servant and base skeleton classes.*

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.

Termination Handler

Orbix provides its own `IT_TerminationHandler` class, which enables server applications to handle delivery of `Ctrl-C` and similar events in a portable manner. On UNIX, the termination handler handles the following signals:

```
SIGINT
SIGTERM
SIGQUIT
```

On Windows, the termination handler is just a wrapper around `SetConsoleCtrlHandler`, which handles delivery of the following control events:

```
CTRL_C_EVENT
CTRL_BREAK_EVENT
CTRL_SHUTDOWN_EVENT
CTRL_LOGOFF_EVENT
CTRL_CLOSE_EVENT
```

You can create only one termination handler object in a program.

In the following example, the main routine creates a termination handler object on the stack. On POSIX platforms, it is critical to create this object in the main thread before creation of any other thread, especially before calling `ORBinit()`. The `IT_TerminationHandler` destructor deregisters the callback, in order to avoid calling it during static destruction.

```
static void
termination_handler_callback(
   long signal
)
{

    cout << "Processing shutdown signal " << signal << endl;
    if (!CORBA::is_nil(orb))
    {
```

```
        cout >> "ORB shutdown ... " << flush;
        orb->shutdown(IT_FALSE);
        cout << "done." << endl;
    }
}

int
main(int argc, char** argv)
{
    IT_TerminationHandler
    termination_handler(termination_handler_callback);
}
```

Compiling and Linking

Server compile and link requirements are almost the same as the client, except that it also requires the server-side skeleton code, which has the format *idl-name*S.cxx—for example, BankDemoS.cxx. You also must link with the poa library, which contains the server-side run-time support for the POA.

Details for compiling and linking a server differ among platforms. For more information about platform-specific compiler flags and libraries, refer to the demo makefiles in your Orbix distribution.

10 Managing Server Objects

A portable object adapter, or POA, provides the mechanism by which a server process maps CORBA objects to language-specific implementations, or servants. 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.

This chapter shows how to create and manage a POA within a server process, covering the following topics:

- Mapping objects to servants
- Creating a POA and setting POA policies
- Using POA policies
- Activating objects implicitly and explicitly
- Using the POA manager to manipulate request flow control

Mapping Objects to Servants

Figure 24 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.

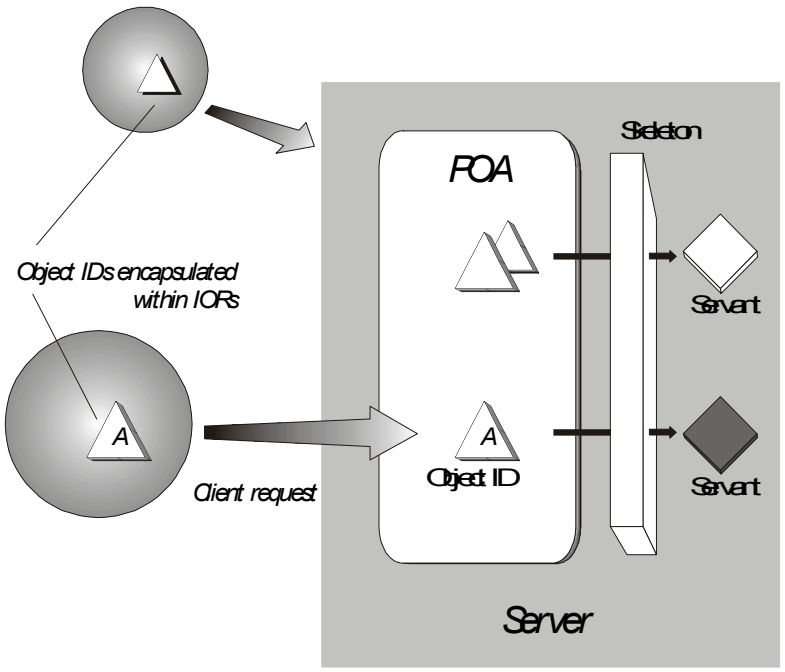

Figure 24: *A portable object adapter (POA) maps abstract objects to their concrete implementations (servants)*

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 228.

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 227). 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.

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 229). 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.

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 13 on page 226).
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. The `PortableServer::POA` interface provides factories to create CORBA Policy object types (see Table 13 on page 226). 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 226). 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`:

```
CORBA::PolicyList policies;
policies.length (2);
policies[0] = poa->create_lifespan_policy
    (PortableServer::PERSISTENT)
```

```
policies[1] = poa->create_id_assignment_policy
    (PortableServer::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.

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
CORBA::Object_var obj =
    orb->resolve_initial_references( "RootPOA" );
PortableServer::POA_var poa = POA::_narrow( obj );

//create policy object
CORBA::PolicyList policies;
policies.length (2);

// set policy object with desired policies
policies[0] = poa->create_lifespan_policy
    (PortableServer::PERSISTENT)
policies[1] = poa->create_id_assignment_policy
    (PortableServer::USER_ID)

//create a POA for persistent objects
poa = poa->create_POA( "persistentPOA", NULL, 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. 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.

Note: Orbix automatically removes policy objects when they are no longer referenced by any POA.

POA Policy Factories

The PortableServer::POA interface contains factory methods for creating CORBA Policy objects:

Table 13: *POA policy factories and argument options*

POA policy factories	Policy options *(d) = default*
create_id_assignment_policy()	SYSTEM_ID *(d)* USER_ID
create_id_uniqueness_policy()	UNIQUE_ID *(d)* MULTIPLE_ID
create_implicit_activation_policy()	NO_IMPLICIT_ACTIVATION *(d)* IMPLICIT_ACTIVATION
create_lifespan_policy()	TRANSIENT *(d)* PERSISTENT
create_request_processing_policy()	USE_ACTIVE_OBJECT_MAP_ONLY *(d)* USE_DEFAULT_SERVANT USE_SERVANT_MANAGER
create_servant_retention_policy()	RETAIN *(d)* NON_RETAIN
create_thread_policy()	ORB_CTRL_MODEL *(d)* SINGLE_THREAD_MODEL

For specific information about these methods, refer to their descriptions in the *Orbix 2000 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:

```
CORBA::PolicyList policies;
policies.length (1);
CORBA::Any obj_deactivation_policy_value;
obj_deactivation_policy_value <<= IT_PortableServer::DISCARD;

policies[0] = orb->create_policy(
    ( IT_PortableServer::OBJECT_DEACTIVATION_POLICY_ID,
     obj_deactivation_policy_value );
```

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 255.
- `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 232.
- `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 232.
- `WorkQueuePolicy`: Allows the application to control the work queue in which incoming requests are placed for processing. For more information, see "Creating a Work Queue" on page 242.

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`

Policy	Default setting
Request Processing	USE_ACTIVE_OBJECT_MAP_ONLY
Servant Retention	RETAIN
Thread	ORB_CTRL_MODEL

Using POA Policies

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.

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 229), 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").

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 228).

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.

A POA with a PERSISTENT policy must be registered in the implementation repository through the `itadmin poa create` command. For more information, see the *Orbix 2000 Administrator's Guide*.

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 for the server process. A POA that uses direct persistence must indicate where the configuration policy sets the well-known address to be embedded in object references. For this purpose, the configuration must contain a well-known address configuration variable, with this syntax:

prefix:*transport*:addr_list=*{...}*

This is done by creating a WellKnownAddressingPolicy object and setting its value to the prefix that contains the well-known address.

For example, you can create a well-known address configuration variable in name scope MyConfigApp as follows:

```
MyConfigApp {
    ...
    wka:IIOP:addr_list=
    ...
}
```

Given this configuration, a POA is created in the ORB MyConfigApp can have its PolicyList set so it generates persistent object references that use direct persistence, as follows:

```
CORBA::PolicyList policies;
policies.length (4);
```

```
CORBA::Any persistence_mode_policy_value;
CORBA::Any well_known_addressing_policy_value;
persistence_mode_policy_value
    <<= IT_PortableServer::DIRECT_PERSISTENCE;
well_known_addressing_policy_value <<=
    CORBA::Any::from_string("MyAppConfigScope", IT_TRUE);

policies[0] = poa->create_lifespan_policy
    (PortableServer::PERSISTENT);
policies[1] = poa->create_id_assignment_policy
    (PortableServer::USER_ID);
policies[2] = orb->create_policy(
    ( IT_PortableServer::PERSISTENCE_MODE_POLICY_ID,
     persistence_mode_policy_value );
policies[3] = orb->create_policy(
    IT_CORBA::WELL_KNOWN_ADDRESSING_POLICY_ID,
    well_known_addressing_policy_value );
```

Object Lifespan and ID Assignment

A POA typically correlates its lifespan and ID assignment policies. `TRANSIENT` and `SYSTEM_ID` are the default settings for a new POA, as system-assigned IDs are generally sufficient for transient object references. `PERSISTENT` and `USER_ID` policies are usually set together, inasmuch as an application typically requires explicit control over the object IDs of its persistent object references.

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.

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.

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 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 Activation" on page 237).

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 "Explicit and Implicit Object Activation" on page 236.

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.

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 242.

Explicit and Implicit Object Activation

A POA's activation policy determines whether a server application can activate objects implicitly or explicitly.

Explicit Activation

If the POA has a 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 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.

`_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.

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_ptr
WhateverImpl::get_self() throw(CORBA::SystemException)
{
    return _this();         // Return reference to self
}
```

Calling _this() Outside an Operation

As discussed earlier, you can call `_this()` on a servant to activate an object. When you do so, the function also returns an object reference. 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:

```
class ServantBase {
public:
    virtual POA_ptr _default_POA();
    // ...
};
```

All skeleton classes and the servants that implement them derive from `ServantBase`, 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.

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 264), 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.

Orbix's cpp_poa_genie.tcl genie uses this approach to override _default_POA() in the servant code that it generates. The genie generates the common base class IT_ServantBaseOverrides, which overrides the definition of _default_POA():

```
//C++
//File: it_servant_base_overrides.h
...
class IT_ServantBaseOverrides :
1       public virtual PortableServer::ServantBase
{
public:
2       IT_ServantBaseOverrides(
            PortableServer::POA_ptr
        );

        virtual
        ~IT_ServantBaseOverrides();

        virtual PortableServer::POA_ptr
3       _default_POA();

private:
4       PortableServer::POA_var m_poa;
        ...
};
```

The code can be explained as follows:

1. IT_ServantBaseOverrides inherits from PortableServer::ServantBase, which is the base class for all servant classes.
2. The constructor is passed a reference to a POA object, which it stores in private member variable m_poa.
3. IT_ServantBaseOverrides::_default_POA() overrides the definition inherited from PortableServer::ServantBase. It returns a copy of the POA reference stored in m_poa.
4. The m_poa private member is used to stores the POA reference.

For more information about using the IT_ServantBaseOverrides class, see page 49.

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.

You can register either an existing POA manager or supply 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.

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 14.

Table 14: *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 14: *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`.

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.

Creating a Work Queue

Orbix provides a proprietary WorkQueue policy, which you can associate with a POA and thereby control the flow of incoming requests for that POA.

A work queue has the following interface definition:

```
interface WorkQueue
{
    readonly attribute long max_size;
    readonly attribute unsigned long count;

    boolean
    enqueue(in WorkItem work, in long timeout);

    boolean
    is_full();

    boolean
    is_empty();

    boolean
    activate();

    boolean
    deactivate();

    void
    flush();
};
```

You can implement your own `WorkQueue` interface, or use IONA-supplied `WorkQueue` factories to create one of two `WorkQueue` types: a `ManualWorkQueue`, or an `AutomaticWorkQueue`.

ManualWorkQueue

A `ManualWorkQueue` is a work queue that holds incoming requests until they are explicitly dequeued. Its interface is defined as follows:

```
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);
};
```

Applications that use a `ManualWorkQueue` must periodically call `dequeue()` or `do_work()` to ensure that requests are handled in a timely manner. A false return value from either method indicates that the timeout has expired or that the queue has shut down.

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
    );
};
```

`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.

AutomaticWorkQueue

An `AutomaticWorkQueue` is a work queue that feeds a thread pool. Its interface is defined as follows:

```
interface AutomaticWorkQueue : WorkQueue {
    attribute long high_water_mark;
    attribute long low_water_mark;

    void
    shutdown(
        in boolean process_remaining_jobs
    );
};
```

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. 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
    );
};
```

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. 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. 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 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.

Creating a POA with a WorkQueue Policy

To create a POA with a WorkQueue policy, follow these steps:

1. Create a work queue factory by calling `resolve_initial_references()` and specify 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.

The following code illustrates these steps:

```
// get an automatic work queue factory
1 CORBA::Object obj_var obj =
      resolve_initial_references("IT_AutomaticWorkQueueFactory");
  IT_WorkQueue::AutomaticWorkQueueFactory_var wqf =
      AutomaticWorkQueueFactory::_narrow( obj );

2 // set work queue parameters
  CORBA::Long max_size = 20;
  CORBA::Long init_thread_count = 1;
  CORBA::Long high_water_mark = 10;
  CORBA::Long low_water_mark = 2;

3 // create work queue
  IT_AutomaticWorkQueue_var auto_wq = wqf->create_work_queue(
      max_size,
      init_thread_count,
      high_water_mark,
      low_water_mark
  );

4 // insert the work queue into an any
  CORBA::Any work_queue_policy_val;
  work_queue_policy_val <<= auto_wq;

  // create PolicyList
  CORBA::PolicyList policies;
  policies.length(3);
```

```
   // other POA policies set
   // ...

5  // add work queue policy object to POA's PolicyList
   policies[0] = orb->create_policy(
       ( IT_WorkQueue::WORK_QUEUE_POLICY_ID,
        work_queue_policy_val);

   // ... add other POA policies to PolicyList
   // ...
```

11 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()`. However, 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. If a POA is required to maintain a large number of objects, you should probably configure it to instantiate servants on demand by setting its request processing policy to `USE_SERVANT_MANAGER`. Or you can set this policy

to USE_DEFAULT_SERVANT to specify 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.

Using Servant Managers

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.

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 POA_BankDemo::Account:

```
// C++
class SingleAccountImpl :
```

```
    public POA_BankDemo::Account
{
  public:
    SingleAccountImpl(
    const char*  account_id,
    AccountDatabase& account_db
    );

    ~SingleAccountImpl();

    void withdraw(BankDemo::CashAmount amount) throw(
        CORBA::SystemException,
        BankDemo::Account::InsufficientFunds);

    void deposit(BankDemo::CashAmount amount) throw(
        CORBA::SystemException);

    char* account_id() throw(CORBA::SystemException);

    BankDemo::CashAmount balance() throw(CORBA::SystemException);

  private:
    CORBA::String_var       m_account_id;
    BankDemo::CashAmount    m_balance;
    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.

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.

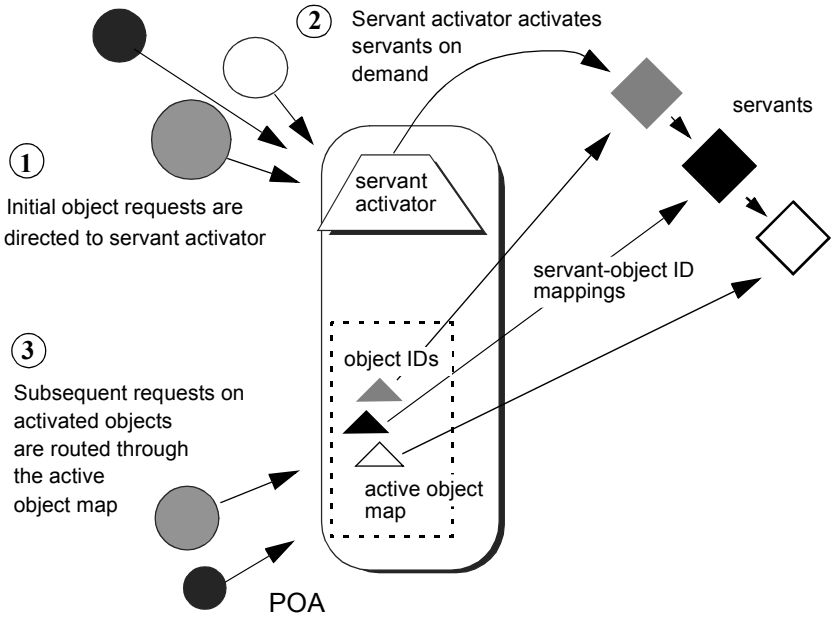

Figure 25: *On the first request on an object, the servant activator returns a servant to the POA, which establishes the mapping in its active object map.*

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 define a servant activator as follows:

```
// C++
#include <omg/PortableServerS.hh>
#include "account_db.h"

class AccountServantActivatorImpl :
    public PortableServer::ServantActivator,
    public CORBA::LocalObject
{
  public:
    AccountServantActivatorImpl(AccountDatabase& account_db);

    PortableServer::Servant incarnate(
    const PortableServer::ObjectId & oid,
    PortableServer::POA_ptr adapter
    ) throw(CORBA::SystemException,
      PortableServer::ForwardRequest);

    void etherealize(
```

```
    const PortableServer::ObjectId & oid,
    PortableServer::POA_ptr adapter,
    PortableServer::Servant serv,
    CORBA::Boolean cleanup_in_progress,
    CORBA::Boolean remaining_activations
    ) throw(CORBA::SystemException);
```

In this example, the servant activator's constructor takes a single argument, an AccountDatabase object, to enable interaction between Account objects and persistent account data..

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 following implementation, `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.

```
// servant activator constructor
AccountServantActivatorImpl::AccountServantActivatorImpl(
    AccountDatabase& account_db) : m_account_db(account_db)
{ // ... }

PortableServer::Servant
1 AccountServantActivatorImpl::incarnate(
    const PortableServer::ObjectId & oid,
    PortableServer::POA_ptr  adapter
) throw(CORBA::SystemException, PortableServer::ForwardRequest)
{
    CORBA::String_var account_id =
        PortableServer::ObjectId_to_string(oid);
2   return new SingleAccountImpl(account_id, m_account_db);
}
```

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.

The following implementation of `etherealize()` checks the `remaining_activations` parameter to ensure that the servant does not incarnate another object before it deletes the servant. Implementations can also 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.

```
void
AccountServantActivatorImpl::etherealize(
    const PortableServer::ObjectId & oid,
    PortableServer::POA_ptr  poa,
    PortableServer::Servant servant,
    CORBA::Boolean cleanup_in_progress,
    CORBA::Boolean remaining_activations
) throw((CORBA::SystemException))
{
    if (remaining_activations == 0)
        delete serv;
}
```

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 226).

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

You establish a POA's servant activator in two steps:

1. Instantiate the servant activator.
2. Call set_servant_manager() on the target POA and supply the servant activator.

```
...
AccountDatabase account_database = new AccountDatabase();

// instantiate servant activator
AccountServantActivatorImpl activator_impl(account_database);
acct_poa->set_servant_manager( &activator_impl );

// Associate the activator with the accounts POA
acct_poa->set_servant_manager( 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.

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.

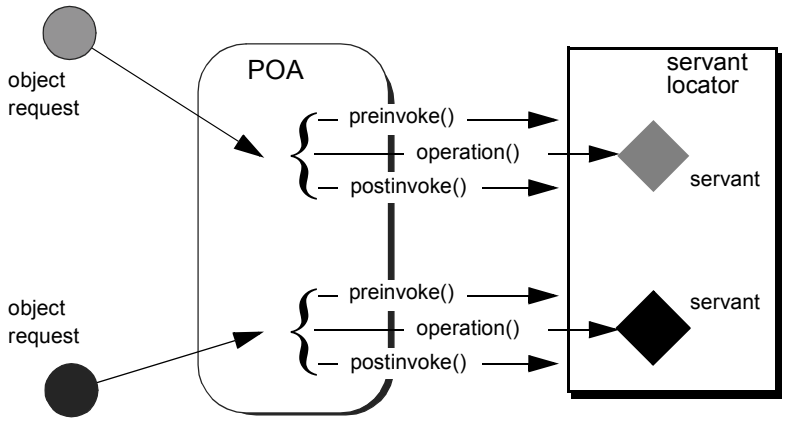

Figure 26: *The POA directs each object request to the servant locator, which returns a servant to the POA to process the request.*

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. The postinvoke() implementation can either delete the servant, or cache it for later reuse.

Implementing a Servant Locator

The following code defines a servant locator that handles account objects:

```
// C++
class AccountServantLocatorImpl :
    public PortableServer::ServantLocator,
    public CORBA::LocalObject
```

```
{
  public:
    AccountServantLocatorImpl(AccountDatabase& account_db);

public:
    PortableServer::Servant preinvoke(
        const PortableServer::ObjectId &id,
        PortableServer::POA_ptr poa,
        const char *operation,
        PortableServer::Cookie &cookie )
        throw( CORBA::SystemException );

    void postinvoke (
        const PortableServer::ObjectId &id,
        PortableServer::POA_ptr poa,
        const char *operation,
        PortableServer::Cookie &cookie,
        PortableServer::Servant the_servant )
        throw(CORBA::SystemException);
```

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.

Incarnating Objects With a Servant Locator

The following implementation of `preinvoke()`is functionally identical to the `incarnate()` implementation shown earlier (see page 254).

```
// C++
PortableServer::Servant
MyAcctLocator::preinvoke(
    const PortableServer::ObjectID &id,
    PortableServer::POA_ptr poa
    const char *operation
    PortableServer::Cookie &cookie )
throw( CORBA::SystemException )
{
    CORBA::String_var str =
        PortableServer::ObjectId_to_string(id);

    // look up account ID in accounts database,
    // make sure it it exists
    CORBA::Long acctId = acct_lookup(str);

    if (acctId == -1)
        throw CORBA::OBJECT_NOT_EXIST ();

    return new SingleAccountImpl(str);
}
```

Etherealizing Objects With a Servant Locator

The following implementation of `postinvoke()` is similar to the `etherealize()` implementation shown earlier (see page 255), with one significant difference: because each servant is bound to a single request, `postinvoke()` has no remaining activations to check.

```
PortableServer::Servant
MyAcctLocator::postinvoke(
    const PortableServer::ObjectID &id,
    PortableServer::POA_ptr poa,
    const char *operation,
    PortableServer::Cookie &cookie,
    PortableServer::Servant the_servant )
```

```
throw( CORBA::SystemException )
{
    delete servant;
}
```

Setting a POA's Servant Locator

You establish a POA's servant locator in two steps:

1. Instantiate the servant locator.
2. Call `set_servant_manager()` on the target POA and supply the servant locator.

```
// C++
AccountServantLocatorImpl locator_impl(account_database);

// Associate the locator with the accounts POA
acct_poa->set_servant_manager( &locator_impl );
```

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.

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.

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:

```
// C++
class SingleAccountImpl : public virtual POA_BankDemo::Account{

public:
    // constructor
    SingleAccountImpl (CORBA::ORB_ptr orb) : orb_ (orb) {}

    // get account holder's name
    char * name() throw(CORBA::SystemException){

        CORBA::String_var acct = get_acct_id();
        // rest of function not shown
    }

    // get account balance
    CORBA::Float balance() throw(CORBA::SystemException){

        CORBA::String_var acct = get_acct_id();
        // rest of function not shown
    }
```

```
    // similar processing for other operations

private:
    char *get_acct_id(void){
        CORBA::Object_var obj =
            orb_->resolve_initial_references("POACurrent");
        PortableServer::Current_var cur =
            PortableServer::Current::_narrow(obj);
        try {
           PortableServer::ObjectID_var id = cur->get_object_id();
            return PortableServer::ObjectID_to_string(id);
        } catch (const PortableServer::Current::NoContext &) {
          cerr << "NoContext error" << endl;
        }
    }
}
        try {
        } catch (org.omg.PortableServer.Current.NoContext) {
          // ...
        }
```

In this implementation, the servant constructor takes a single argument, a pointer 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 (`PortableServer::ObjectID_to_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 264 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`:

```
// C++
// 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 );
```

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:

```
// C++
int main( int argc, char **argv) {
    // initialize ORB
    CORBA::ORB_var orb = CORBA::ORB_init( argc, argv );

    // get object reference to the root POA
    CORBA::Object_var obj =
```

```
    orb->resolve_initial_references( "RootPOA" );
PortableServer::POA_var poa = POA::_narrow( obj );

// set policies for persistent POA that uses servant locator
CORBA::PolicyList policies;
policies.length (2);
policies[0] = poa–>create_lifespan_policy
    (PortableServer::PERSISTENT)
policies[1] = poa–>create_id_assignment_policy
    ( PortableServer::USER_ID )
policies[2] = poa–>create_servant_retention_policy
    ( PortableServer::NON_RETAIN )
policies[3] = poa–>create_request_processing_policy
    ( PortableServer::USE_SERVANT_MANAGER )

// create the POA
poa = poa->create_POA( "acct_poa", NULL, policies );

AccountDatabase account_database = new AccountDatabase();

AccountServantLocatorImpl locator_impl(account_database);

// Associate the locator with the accounts POA
acct_poa->set_servant_manager( &locator_impl );

// Set Bank Account interface repository ID
const char *repository_id = "IDL:BankDemo/Account:1.0";

// create account object
PortableServer::ObjectId_var acct_id =
    PortableServer::string_to_ObjectId( "112-1110001");
CORBA::Object_var acctObj =
    acct_poa->create_reference_with_id(
        acct_id, repository_id);

// Export object reference to Naming Service (not shown)

// create another account object
PortableServer::ObjectId_var acct_id =
    PortableServer::string_to_ObjectId( "112-1110002");
CORBA::Object_var acctObj =
    acct_poa->create_reference_with_id(
        acct_id, 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.

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.

12 Asynchronous Method Invocations

Orbix support for asynchronous method invocations allows a client to continue other work while it awaits responses from previous requests.

Examples of client implementations in earlier chapters show client invocations that follow a synchronous two-way model—that is, after a client sends a request, it blocks on that thread until it receives a reply. If single-threaded, the client is generally unable to perform any other work while it awaits a response. This can be unacceptable in an application that requires clients to issue requests in rapid succession and needs to process replies as soon as they become available.

To avoid this problem, Orbix supports asynchronous method invocations (AMI) through callbacks to reply handlers. In its invocation, the client supplies an object reference to the appropriate reply handler. When it is ready to reply, the server invokes on this object reference. The client ORB dispatches the invocation to the reply handler servant

In most cases, AMI usage affects only client implementations; servers are unaware that an invocation is synchronous or asynchronous. Client asynchrony matters only to transactional servers, and in this case can require changes to the server.

The examples in this chapter use the following IDL, which queries banking institutions for current lending rates:

```
module LoanSearch
{
    // nonexistent Bank
    exception InvalidBank{};
    // invalid loan type
    exception InvalidLoanType{};

    interface LoanRates{
```

```
        float get_loan_rate(
            in string bank_name,
            in string loan_type
        ) raises (InvalidBank, InvalidLoanType);
    };
    // ...
};
```

Client implementations must be able to invoke the `get_loan_rate()` operation asynchronously on multiple lenders, so that information from each one can be reviewed as soon as it is available, without waiting for previous queries to return. Each implementation uses the following global variables:

```
static const char *banks[] =
{
    "Fleet",
    "Citizens",
    "BkBoston",
    "USTrust",
    //...
}
static const int MAX_BANKS = sizeof(banks);
static const int replies_left = MAX_BANKS;

static const char *loan_types[] =
{
    "AUTO",
    "MORTGAGE",
    "EQUITY",
    "PERSONAL",
    "BUSINESS",
    // ...
}
```

Implied IDL

In order to support AMI, the IDL compiler provides the `-xAMICallbacks` option. This generates an *implied IDL* sendc_ operation for each interface operation and attribute, which supports AMI callbacks. You must supply the `-xAMICallbacks` modifer with both `-base` and `-poa` switches, as in the following example:

```
IDL -poa:-xAMICallbacks -base:-xAMICAllbacks LoanSearch.idl
```

For example, given the `get_loan_rate()` operation, the IDL compiler generates an implied IDL `sendc_get_loan_rate()` operation that it adds to the `LoanRates` interface. The compiler then generates stub and skeleton code from the entire set of explicit and implicit IDL.

Mapping Operations to Implied IDL

In general, each `in` and `inout` parameter in an IDL operation is mapped to an `in` parameter of the same name and type in the corresponding implied IDL operation. `sendc_` operations return void and supply as their first argument an object reference to the client-implemented reply handler. They have the following syntax

```
void sendc_op-name(
    reply-hdlr-ref,
    [ in type argument[, in type argument]... );
```

Mapping Attributes to Implied IDL

Each IDL attribute is mapped to a `sendc_get_` operation. If the attribute is not read-only, the IDL compiler also generates a `sendc_set_` operation, which has a single `in` parameter of the same name and type as the attribute.

`sendc_get_` and `sendc_set_` operations return void and supply as their first argument an object reference to the client-implemented reply handler. They have the following syntax:

```
void sendc_get_attribute-name( reply-hdlr-ref );
void sendc_set_attribute-name(
    reply-hdlr-ref,
    in type attribute-name);
```

Calling Back to Reply Handlers

For each IDL operation and attribute, the IDL compiler generates:

- A sendc_ operation that supports AMI callbacks.
- A reply handler class for each interface, derived from `Messaging::ReplyHandler`.

The generated reply handler class name uses the following convention:

`AMI_`*`interface-name`*`Handler`

If the generated handler name conflicts with other IDL definitions, the ORB prepends additional strings of `AMI_` until the name is unique. For example, all send_c invocations on interface `LoanRates` take a reference to an instance of `AMI_LoanRatesHandler` as their first argument.

The client instantiates reply handlers like any servant, and registers them with a client-side POA. If a reply handler serves time-independent invocations, its object reference must be persistent.

For each sendc_ invocation on the interface, the following events occur:

1. The client supplies an object reference to the invocation's reply handler.
2. The invocation returns immediately to the client, which can continue processing other tasks while it awaits a reply.
3. The server invokes on the reply handler when a reply is ready.

Note: A client-side POA has the same requirements as a POA that is implemented on a server—for example, the POAManager must be in an active state before the client can process reply handler callbacks.

Interface-to-Reply Handler Mapping

The client can implement a reply handler for each interface. For each interface operation and attribute, a reply handler provides two types of operations: one to handle normal replies, and another for exceptional replies.

For example, when you run the IDL compiler on interface LoanSearch::LoanRates (shown earlier), it generates skeleton class POA_LoanSearch::AMI_LoanRatesHandler:

```
namespace POA_LoanSearch{
    class AMI_LoanRatesHandler
        : public POA_Messaging::ReplyHandler{
    public:
        // ...
        virtual void
        get_loan_rate_complete(
            CORBA::Float ami_return_val
        ) IT_THROW_DECL((CORBA::SystemException)) = 0;

       // ...
        virtual void
        get_loan_rate_excep(
            Messaging::ExceptionHolder*  ami_holder
        ) IT_THROW_DECL((CORBA::SystemException)) = 0;
    };
}
```

LoanRates contains only one operation, LoanRates::get_loan_rate(), which maps to AMI operation sendc_get_loan_rate(). The reply handler AMI_LoanRatesHandler therefore has two operations:

- get_loan_rate_complete() handles normal replies to sendc_get_loan_rate().
- get_loan_rate_excep() handles exceptions that might be raised by sendc_get_loan_rate().

So, if the client invokes sendc_get_loan_loan_rate() and supplies a valid bank name and loan type, the client ORB invokes an implementation of AMI_LoanRatesHandler::get_loan_rate_complete() to handle the reply. However, if either argument is invalid, the client ORB invokes AMI_LoanRatesHandler::get_loan_rate_excep().

Normal Replies

A reply handler can contain up to three types of operations to handle normal replies—that is, replies on invocations that raise no exceptions:

Table 15: *Reply Handler Operation Types for Normal Replies*

For invocations on...	The reply handler uses...

Operation	An operation with the same name:
	`void op-name_complete(`
	`[,in type ami_return_val`
	`[,in type argument]...`
	`);`

Read-only attribute	A `get_` operation: `void get_attr-name(in type ami_return_val);`
Read/write attribute	A `set_` operation: `void set_attr-name();`

In general, an `in` argument is included for each `out` or `inout` parameter in the IDL definition. All arguments have the same type as the original IDL. If the invocation returns a value, the first argument contains that value; otherwise, arguments have the same order as in the original IDL.

Exceptional Replies

A reply handler can contain up to three types of operations to handle exceptional replies:

Table 16: *Reply Handler Operation Types for Exceptional Replies*

For invocations on...	The reply handler uses...
Operation	`void op-name_excep( in Messaging::ExceptionHolder ami_holder);`
Read-only attribute	`void get_attr-name_excep( in Messaging::ExceptionHolder ami_holder);`
Read/write attribute	`void set_attr-name_excep( in Messaging::ExceptionHolder ami_holder);`

All three operations contain a single `in` argument of type `Messaging::ExceptionHolder`, which contains the exception raised by the original client invocation. You access this exception using `get_exception()`. The call returns an `Any*` from which the exception can be extracted.

Implementing a Client with Reply Handlers

As shown earlier, the reply handler `AMI_LoanRatesHandler` for interface `LoanRates` contains two operations to handle normal and exceptional replies to `sendc_get_loan_rate()`. The client implementation of this reply handler might look like this:

```
class MyLoanRatesHandler :
    LoanRates::AMI_LoanRatesHandler{
public:
    // handler constructor
    MyLoanRatesHandler(const char *bank_name, loan_type) :
        bank_name_(CORBA::string_dup(bank_name),
        loan_type_(CORBA::string_dup(loan_type))
    { }
    ~MyLoanRatesHandler(void)
```

```
{ }

// process normal replies
virtual void get_loan_rate_complete(CORBA::Float reply_val)
    throw(CORBA::SystemException)
    {
    cout << loan_type_
         << "loan: from "
         << bank_name_
         << "  Current rate is "
         << reply_val;

    // Decrement the number of replies still pending
    replies_left--;
    }

// process exceptional replies
virtual void get_loan_rate_excep(
    Messaging::ExceptionHolder* ami_holder)
    throw(CORBA::SystemException, LoanRates::InvalidBank,
        LoanRates::InvalidLoanType)
   {
     CORBA::Any* tmp = ami_holder->get_exception();
     if(LoanSearch::IT_Gen_InvalidBankStreamable::
extract_from(tmp))   {
         cerr << bank_name_
         cerr << " is not a valid bank name."
         throw(LoanRates::InvalidBank);
     }
     else if(LoanSearch::IT_Gen_InvalidLoanStreamable::
extract_from(tmp)) {
         cerr << loan_type_
         cerr << " is not a valid loan type."
         throw(LoanRates::InvalidBank);
     }
     else {
         cerr << "get_loan_rate() raised exception "
              << tmp
              << " for "
              << bank_name_
              << " and "
              << bank_type_
          throw(CORBA:SystemException tmp);
```

```
        }
        // Decrement the number of replies still pending
        replies_left--;
    }

private:
    CORBA::String_var bank_name_, bank_type_ ;
}
```

In the following client implementation, the client performs these actions:

1. Calls `get_latest_rates()` and supplies it with three arguments: a pointer to the client ORB, an object reference to the `LoanSearch` object, and the desired loan type.
2. Calls the callback operation `sendc_get_loan_rates()` repeatedly, once for each bank. Each call to `sendc_get_loan_rates()` supplies an `AMI_LoanRatesHandler` reply handler argument:

```
void get_latest_rates(
    CORBA::ORB_ptr,
    LoanSearch::LoanRates_ref,
    CORBA::String loan_type)
{
    // array of pointers to bank reply handlers
    MyLoanRatesHandler *handlers[MAX_BANKS];

    // create object references for each reply handler
    LoanRates::AMI_LoanRatesHandler_ptr *handler_refs[MAX_BANKS];

    int i;

    // instantiate reply handler servants
    for(i = 0; i < MAX_BANKS; i++)
        handlers[i] = new MyLoanRatesHandler(
            banks[i], loan_types[i]);

    // get object references to reply handlers
    for(i = 0; i < MAX_BANKS; i++)
        handler_refs[i] = handlers[i]->_this();

    // Issue asynchronous calls via callbacks
    for(i = 0; i < MAX_BANKS; i++)
        LoanRates_ref->sendc_get_loan_rates(
```

```
            handler_refs[i], banks[i], loan_type);
```

After all synchronous calls are invoked, the client can await replies within the ORB's event loop:

```
    // iterate within ORB event loop until all replies
    // are processed
    while(replies_left > 0)
        if(orb->work_pending())
            orb->perform_work();
}
```

13 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.

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);
    };
};
```

Exception Code Mapping

The C++ mapping arranges CORBA exceptions into the hierarchy shown in Figure 27. Abstract base class `CORBA::Exception` is the root of the hierarchy tree. Base abstract classes `SystemException` and `UserException` derive from `CORBA::Exception` and provide the base for all concrete system and user exceptions:

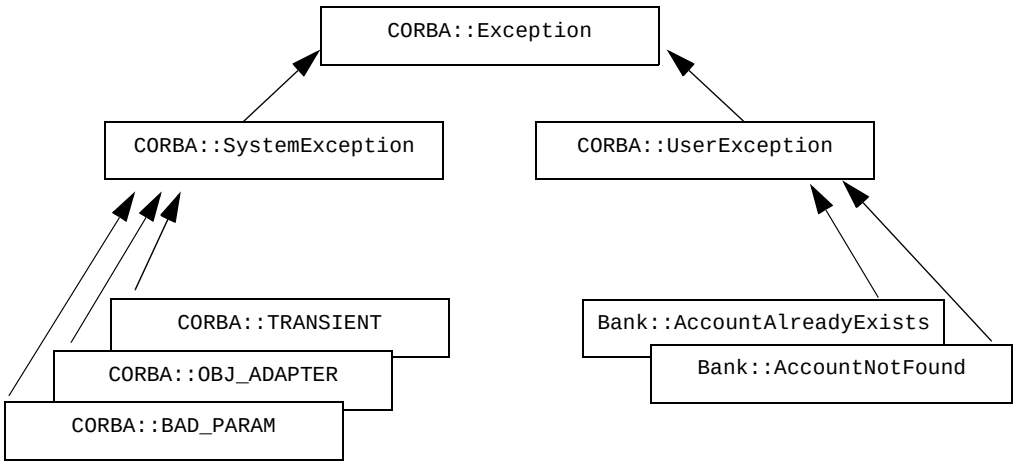

Figure 27: *The C++ mapping arranges exceptions into a hierarchy*

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.

C++ Mapping for User Exceptions

When you run the IDL compiler on IDL interface `Bank`, it translates user exceptions into C++ classes. For example, the compiler translates `Bank::AccountAlreadyExists` into a C++ class of the same name:

```
class Bank : public virtual CORBA::Object
{
public:
...
    class AccountAlreadyExists: public CORBA::UserException
    {
    public:

        AccountAlreadyExists();
        AccountAlreadyExists(const char* _itfld_account_id);
        ...
        // string manager
        ITGenAccountId_mgr account_id;

        static AccountAlreadyExists* _downcast(
            CORBA::Exception* exc
        );
        static const AccountAlreadyExists* _downcast(
            const CORBA::Exception* exc
        );
        ...
        virtual void _raise() const;
        ...
    };
    ...
};
```

The `AccountAlreadyExists` class is nested within class `Bank`. Each C++ class that corresponds to a IDL exception has a constructor that takes a parameter for each exception member. Because the `AccountAlreadyExists` exception has one `AccountId` member, class `Bank::AccountAlreadyExists` has a constructor that allows it to be initialized.

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.

User Exceptions

If an operation might throw a user exception, its caller should be prepared to handle that exception with an appropriate `catch` clause.

The following code 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. The code follows standard C++ practice by passing the parameter to the `catch` clause by reference. The `operator<<()` that is defined on class `SystemException` outputs a text description of the individual system exception that was thrown.

```
// C++
void
BankMenu::do_create()
    throw(CORBA::SystemException)
{
    cout << "Enter account name: " << flush;
    char name[1024];
    cin  >> name;
    cout << "Enter starting balance: " << flush;
    BankDemo::CashAmount amount;
    cin  >> amount;

    // try/catch to handle user exception, system exceptions are
    // handled in the main menu loop
    try
    {
        BankDemo::Account_var account =
            m_bank->create_account(name, amount);

        // start a sub-menu with the returned account reference
        AccountMenu sub_menu(account);
        sub_menu.run();
```

```
        // _var types automatically clean up on return
        // or exception
    }
    catch (
      const BankDemo::Bank::AccountAlreadyExists& already_exists)
    {
        cout << "Account already exists: "
             << already_exists.account_id << endl;
    }
}
```

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.

The handler for a specific system exception must appear before the handler for `CORBA::SystemException`. C++ `catch` clauses are attempted in the order specified, and the first matching handler is called. Because of implicit casting, a handler for `CORBA::SystemException` matches all system exceptions (all system exception classes are derived from class `CORBA::SystemException`), so it should appear after all handlers for specific system exceptions.

If you want to know the type of system exception that occurred, use the message output by the proprietary `operator<<()` function on class `CORBA::SystemException`. Handlers for individual system exceptions are necessary only when they require a specific action.

The following client code specifically tests for a `COMM_FAILURE` exception; it can also handle any other system exceptions:

```
void
BankMenu::run() {
    // make sure bank reference is valid
    if (CORBA::is_nil(m_bank)) {
        cout << "Cannot proceed - bank reference is nil";
    }
    else {
    // loop printing the menu and executing selections
```

```
            for ( ; ; ) {
                cout << endl;
                cout << "0 - quit" << endl;
                cout << "1 - create_account" << endl;
                cout << "2 - find_account" << endl;
                cout << "Selection [0-2]: " << flush;
                int selection;
                cin >> selection;

                try {
                    switch(selection)
                    {
                        case 0: return;
                        case 1: do_create(); break;
                        case 2: do_find(); break;
                    }
                    catch (CORBA::COMM_FAILURE& e) {
                    cout << "Communication failure exception: "
                         << e << endl;
                    return;
                }
                catch (const CORBA::SystemException& e) {
                    cout << "Unexpected exception: " << e << endl;
                    return;
                }
            }
        }
    }
```

Evaluating System Exceptions

System exceptions have two member methods, `completed()` and `minor()`, that let a client evaluate the status of an invocation:

- `completed()` returns an enumerator that indicates how far the operation or attribute call progressed.
- `minor()` returns an IDL `unsigned long` that 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

Each ORB vendor has a unique VMCID assigned by the OMG. The VMCID assigned to IONA is `0x49540000`; this space is reserved for use by IONA exception minor codes.

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.

Orbix 2000 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 17 lists Orbix 2000 subsystems and base values for their minor codes:

Table 17: *Base minor code values for Orbix subsystems*

Subsystem Base	Minor Code ID
Core	`0x49540100`
GIOP	`0x49540200`
IIOP	`0x49540300`
IIOP_PROFILE	`0x49540400`
POA	`0x49540500`
PSS	`0x49540600`
DAL_DB	`0x49540700`
PSS	`0x49540800`
OTS	`0x49540900`
OTS_LITE	`0x49540A00`
Locator	`0x49540B00`
POA locator	`0x49540C00`
Activator	`0x49540D00`
Generic server	`0x49540E00`
Naming	`0x49540F00`
IFR	`0x49541000`

Table 17: *Base minor code values for Orbix subsystems*

Subsystem Base	Minor Code ID
Configuration repository.	0x49541100
Threads package	0x49541200
PSS/R ODBC	0x49541300
ATLI-IOP none	0x49541400

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, and are defined as follows:

```
// IDL: in location_minor_codes.idl
module IT_LOCATOR_MinorCodes
{
    const unsigned long SMCID = IT_ErrorCodes::IONA_VMCID + 0x0B00;

    module BAD_PARAM
    {
        const unsigned long NO_ACTIVATOR_NAME       = SMCID;
        const unsigned long NO_ACTIVATOR_INFO       = SMCID + 1;
        const unsigned long ACTIVATOR_REG_NO_NAME   = SMCID + 2;
        const unsigned long ACTIVATOR_REG_NO_REF    = SMCID + 3;
        const unsigned long ACTIVATOR_UNREG_NO_NAME = SMCID + 4;
        const unsigned long UNEXPECTED_NULL         = SMCID + 5;
        const unsigned long PROCESS_NOT_EXIST       = SMCID + 6;
    };
    ...
};
```

These equate to the following minor code IDs:

Table 18: *BAD_PARAM minor codes*

Minor code string	Minor code ID
NO_ACTIVATOR_NAME	0x49540B00
NO_ACTIVATOR_INFO	0x49540B01
ACTIVATOR_REG_NO_NAME	0x49540B02
ACTIVATOR_REG_NO_REF	0x49540B03
ACTIVATOR_UNREG_NO_NAME	0x49540B04
UNEXPECTED_NULL	0x49540B05
PROCESS_NOT_EXIST	0x49540B06

For example, an exception with a minor code of `0x49540B06` indicates that the locator is looking for a process that does not exist.

Definitions for all subsystem minor codes can be found in the `idl/orbix_sys` directory.

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.

Throwing Exceptions

Client code uses standard C++ 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:

```
//  create a new account given an id and initial balance
//  throw AccountAlreadyExists if account already in database
```

```
BankDemo::Account_ptr BankImpl::create_account(
    const char* account_id,
    CashAmount initial_balance) throw(
    CORBA::SystemException, BankDemo::Bank::AccountAlreadyExists)
{
    // create new account in database, then return a new
    // reference to that account
    if (!m_account_db.create_account(account_id, initial_balance))
    {
        throw BankDemo::Bank::AccountAlreadyExists(account_id);
    }

    return create_account_ref(account_id);
}
```

Exception Safety

You should be careful that your code does not throw user exceptions that are not part of the operation's raises expression. Doing so can throw an `UNKNOWN` exception, or cause the program to terminate abruptly.

For example, the following IDL defines operations `some_operation()` and `some_helper()`:

```
exception Failed {};
interface Example {
    void some_operation() raises(Failed);
};

exception DidntWork {};
interface Helper {
    void some_helper() raises(Failed, DidntWork);
};
```

The following implementation of `some_operation()` incorrectly calls `some_helper()`:

```
void ExampleImpl::some_operation()
    throw(CORBA::SystemException, Failed) {
    // do some work...
    // call helper operation.
```

```
    Helper_var help = ...;
    help->some_helper(); // BAD!
    // do remainder of work...
}
```

At some point during runtime, `some_helper()` is liable to throw an exception of `DidntWork` back to `some_operation()`, which is unable to handle it, and causing the server process to die.

If an operation calls helper operations on other objects, make sure that it can handle illegal exceptions. For example, the following example modifies `some_operation()` so that it can translate `DidntWork` into a legal exception:

```
void ExampleImpl::some_operation()
    throw(CORBA::SystemException, Failed) {
    // do some work...
    // call helper operation.
    Helper_var help = ...;
    try {
        help->some_helper();
    }
    catch (const DidntWork &) {
        throw Failed; // translate into legal exception
    }
    // do remainder of work...
    return;
}
```

You should also be careful to avoid resource leaks in the presence of exceptions. For example, the IDL for `some_operation()` is modified here to return a string as an out parameter:

```
exception Failed {};
interface Example {
    void some_operation(out string s) raises(Failed);
};
```

The following implementation incorrectly leaks the string that is allocated to the `out` parameter:

```
void ExampleImpl::some_operation(CORBA::String_out s)
    throw(CORBA::SystemException, Failed) {

    // do some work to get the string value to be returned...
    char * str = some_function();
```

```
    s = CORBA::string_dup(str);   // assign out param

    // call helper operation to do something else
    Helper_var help = ...;
    try {
        help->some_helper();      // memory leak!
    }
    catch (const DidntWork &) {
        throw Failed;             // memory leak!
    }
    // do remainder of work...
}
```

You can correct this problem by explicitly deallocating the parameter again, as in the following example:

```
void ExampleImpl::some_operation(CORBA::String_out s)
    throw(CORBA::SystemException, Failed) {

    // do some work to get the string value to be returned...
    char * str = some_function();
    s = CORBA::string_dup(str); // assign out param

    // call helper operation to do something else
    Helper_var help = ...;
    try {
        help->some_helper();
    }
    catch (const DidntWork &) {
        CORBA::string_free(s.ptr());  // clean up
        throw Failed; // translate
    }
    catch (const CORBA::Exception & e) {
        CORBA::string_free(s.ptr());  // clean up
        throw;                        // rethrow
    }
    // do remainder of work...
}
```

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:

```
// C++
class SystemException : public Exception {
    public:
        SystemException();
        SystemException(const SystemException &);
        SystemException(
            ULong minor_id, CompletionStatus completed_status);

class COMM_FAILURE : public SystemException { ... };
```

The following code uses this constructor to throw a COMM_FAILURE exception with minor code SOCKET_WRITE_FAILED and completion status COMPLETED_NO:

```
// C++
throw CORBA::COMM_FAILURE(HOST_LOOKUP_FAILED, COMPLETED_NO);
```

14 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.

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 19 shows type code parameters. The table omits type codes with an empty parameter list.

Table 19: *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]
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. Some of these can be invoked on all TypeCode objects; the rest 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:

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—for example, use the TCKind return as a switch discriminator:

```
//C++
CORBA::Any another_any = ...;
CORBA::TypeCode_var t = another_any.type();

switch(t->kind()){
case CORBA::tk_short:
...
case CORBA::tk_long:
...
// continue for all tk_ values
default:
...
}
```

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.

Both equal() and equivalent() can take a type code constant as an argument—for example, _tc_short or _tc_float for IDL types short or float respectively. For more information about type code constants, see page 301.

You must use equal() and equivalent() to evaluate a type code. For example, the following code is illegal:

```
CORBA::Any another_any;
another_any <<= "Hello world";
CORBA::TypeCode_ptr t = another_any.type();

if (t == CORBA::_tc_string) { ... } // Bad code!!
```

You can correct this code as follows:

```
CORBA::Any another_any;
another_any <<= "Hello world";
CORBA::TypeCode_ptr t = another_any.type();

// use equal or equivalent to evaluate type code
if (t->equivalent(CORBA::_tc_string)) { ... }
if (t->equal(CORBA::_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.

Type-Specific Operations

Table 20 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 20: *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()`

`tk_fixed`	`fixed_digits()` `fixed_scale()`
`tk_native`	`id()` `name()`

Table 20: *Type-Specific Operations*

TCKind	Operations
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 21 briefly describes the information that you can access through type code-specific operations. For detailed information about these operations, see the *Orbix 2000 Programmer's Reference*.

Table 21: *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 21: *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 Code Constants

Orbix provides type code constants that you can use to evaluate and compare type code objects. Built-in type code constants are provided for each `TCKind` enumerator (see page 293). The IDL compiler also generates type code constants for IDL types that you declare in your application code.

Built-In Type Codes

Orbix provides predefined `CORBA::TypeCode` object reference constants that let you access type codes for standard types.

```
CORBA::_tc_any
CORBA::_tc_boolean
CORBA::_tc_char
CORBA::_tc_double
CORBA::_tc_float
CORBA::_tc_long
CORBA::_tc_longdouble
CORBA::_tc_longlong
CORBA::_tc_null
CORBA::_tc_octet
CORBA::_tc_short
CORBA::_tc_string
CORBA::_tc_ulong
CORBA::_tc_ulonglong
CORBA::_tc_ushort
CORBA::_tc_void
CORBA::_tc_wchar
CORBA::_tc_wstring
CORBA::_tc_Object
CORBA::_tc_TypeCode
CORBA::_tc_ValueBase
```

User-Defined Type Codes

The IDL compiler generates type code constants for declarations of these types:

```
interface
typedef
struct
union
enum
valuetype
valuebox
```

For each user-defined type that is declared in an IDL file, the IDL compiler generates a `CORBA::TypeCode_ptr` that points to a type code constant. These constants have the format `_tc_`*`type`* where *`type`* is the user-defined type. For example, given the following IDL:

```
interface Interesting {
    typedef long longType;
    struct Useful
    {
        longType l;
    };
};
```

the IDL compiler generates the following `CORBA::TypeCode_ptr` constants:

```
_tc_Interesting
Interesting::_tc_longType
Interesting::_tc_Useful
```

15 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).

The IDL any type maps to the C++ `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.

This chapter covers the following topics:

- Inserting values into an any data type.
- Querying an any data type for its data.
- Using DynAny objects to construct and interpret any data types dynamically.

Inserting Typed Values Into Any

The insertion operator <<= lets you set an any's value and data type. The insertion operator sets a `CORBA::Any` value and its data type property (`CORBA::TypeCode`). Thus set, you can extract an any's value and data type through the corresponding extraction operator (see page 306).

The C++ class `CORBA::Any` contains predefined overloaded versions of the insertion operator function `operator<<=()`. Orbix provides insertion operator functions for all IDL types that map unambiguously to C++ types, such as `long`, `float`, or unbounded `string`. For a full listing of these functions and their data types, refer to `CORBA::Any::operator<<=()`. The IDL compiler also generates an insertion operator for each user-defined type.

For example, `CORBA::Any` contains the following insertion operator function for `short` data types:

```
void operator<<=(CORBA::Short s);
```

Given this function, you can use the insertion operator to supply a short data type to `passSomethingIn()` as follows:

```
//C++
void AnyDemo::do_send_short() {
    try {
        AnyDemo_var x = ...;
        CORBA::Any a;
        CORBA::Short toPass;
        toPass = 26;
        a <<= toPass;
        x->passSomethingIn(a);
    }
    catch (CORBA::SystemException &sysEx) {
    ...
}
```

Insertion operators provide a type-safe mechanism for inserting data into an any. The type of value to insert determines which insertion operator is used. Attempts to insert a value that has no corresponding IDL type yield compile-time errors.

Memory Management of Inserted Data

Depending on the type of the data, insertion using an `operator<<=()` has one of the following effects:

- `_duplicate()` is called on an object reference.
- `_add_ref()` is called on a valuetype.
- a deep copy is made for all other data types.

When the `Any` is subsequently destroyed, the `Any` destructor performs one of the following actions, depending on the `Any.type()` field:

- `CORBA::release()` is called on an object reference.
- `_remove_ref()` is called on a valuetype.
- `delete` is called on all other data types.

Inserting User-Defined Types

The IDL shown earlier can be modified to include this `typedef` declaration:

```
// IDL
typedef sequence<long> LongSequence;
```

Given this statement, the IDL compiler generates the following insertion operator function for `LongSequence` data types:

```
void operator<<=(CORBA::Any& a, const LongSequence& t);
```

Clients that call `passSomethingIn()` can use the insertion operator to insert `LongSequence` data into the function's any parameter:

```
// C++
void AnyDemo::do_send_sequence() {
    try {
        CORBA::Any a;

        // Build a sequence of length 2
        LongSequence sequence_to_insert(2);
```

```
            sequence_to_insert.length(2);

            // Initialize the sequence values
            sequence_to_insert[0] = 1;
            sequence_to_insert[1] = 2;

            // Insert sequence into the any
            a <<= sequence_to_insert;
            ...
            // Call passSomethingIn and supply any data as argument
            m_any_demo->passSomethingIn (a);
        }
        catch (CORBA::SystemException &sysEx) {
        ...
        }
    }
```

Extracting Typed Values From Any

The extraction operator >>= lets you get the value that a `CORBA::Any` contains and returns a `CORBA::Boolean`: true (`1`) if the any's TypeCode matches the extraction operation's target operand, or false (`0`) if a mismatch occurs.

The C++ class `CORBA::Any` contains predefined overloaded versions of the extraction operator function `operator>>=()`. Orbix provides extraction operator functions for all IDL types that map unambiguously to C++ types, such as `long`, `float`, or unbounded `string`. For a full listing of these functions and their data types, refer to `CORBA::Any::operator>>=()`. The IDL compiler also generates an extraction operator for each user-defined type.

For example, `CORBA::Any` contains the following extraction operator function for `short` data types:

```
//C++
CORBA::Boolean operator>>=(CORBA::Short& s) const;
```

Given this function, a server implementation of `passSomethingIn()` can use the extraction operator to extract a `short` from the function's parameter `anyIn`:

```
// C++
void AnyDemo_i::passSomethingIn(const CORBA::Any& anyIn ) {

    CORBA::Short toExtract = 0;

    if (anyIn >>= toExtract) {
        // Print the value
        cout << "passSomethingIn() returned a string:"
            << toExtract << endl << endl;
    }
    else {
        cerr << "Unexpected value contained in any" << endl;
    }
}
```

Memory Management of Extracted Data

When a user-defined type is extracted from an Any, the data is not copied or duplicated in any way. The extracted data is, therefore, subject to the following restrictions:

- No modifications to the extracted data are allowed. The extracted data is read-only.
- Deallocation of the extracted data is not allowed. The Any retains ownership of the data.

To overcome the restrictions on extracted data, you must explicitly make a copy of the data and modify the new copy instead.

Extracting User-Defined Types

More complex, user-defined types can be extracted with the extraction operators generated by the IDL compiler. For example, the IDL shown earlier can be modified to include this typedef declaration:

```
// IDL
typedef sequence<long> LongSequence;
```

Given this statement, the IDL compiler generates the following extraction operator function for LongSequence data types:

```
CORBA::Boolean operator>>=
    (CORBA::Any& a, LongSequence*& t) const;
```

The generated extraction operator for user-defined types takes a pointer to the generated type as the second parameter. If the call to the operator succeeds, this pointer points to the memory managed by the CORBA::Any. Because a CORBA::Any manages this memory, it is not appropriate to extract its value into a _var variable—attempting to do so results in a compile-time error.

You can extract a LongSequence from a CORBA::Any as follows:

```
void AnyDemo::do_get_any() {
    CORBA::Any_var a;
    cout << "Call getSomethingBack" << endl;
    a = m_any_demo->getSomethingBack();

    LongSequence* extracted_sequence = 0;

    if (a >>= extracted_sequence) {
        cout << "returned any contains sequence with value :"
             << endl;
        print_sequence(extracted_sequence);
    }

    else {
        cout << "unexpected value contained in any" << endl;
    }
}
```

Note: It is an error to attempt to access the storage associated with a CORBA::Any after the CORBA::Any variable has been deallocated.

Inserting and Extracting Booleans, Octets, Chars and WChars

Orbix's IDL to C++ mapping for IDL types `char`, `wchar`, `boolean` and `octet` prevents the overloaded insertion and extraction operators from distinguishing between these four data types. Consequently, you cannot use these operators directly to insert and extract data for these three IDL types.

The `CORBA::Any` class contains a set of insertion and extraction operator functions that use helper types for `char`, `wchar`, `boolean`, and `octet` types:

```
void operator<<=(CORBA::Any::from_char c);
void operator<<=(CORBA::Any::from_wchar wc);
void operator<<=(CORBA::Any::from_boolean b);
void operator<<=(CORBA::Any::from_octet o);

Boolean operator>>=(CORBA::Any::to_char c) const;
Boolean operator>>=(CORBA::Any::to_wchar wc) const;
Boolean operator>>=(CORBA::Any::to_boolean b) const;
Boolean operator>>=(CORBA::Any::to_octet o) const;
```

You can use these helper types as in the following example:

```
// C++
CORBA::Any a;

// Insert a boolean into CORBA::Any a
CORBA::Boolean b = 1;
a <<= CORBA::Any::from_boolean(b);

// Extract the boolean
CORBA::Boolean extractedValue;
if (a >>= CORBA::Any::to_boolean(extractedValue)){
    cout << "Success!" << endl;
}
```

Inserting and Extracting Array Data

IDL arrays map to regular C++ arrays. Because arrays can have different lengths and an array variable points only to the array's first element, the IDL compiler generates a distinct C++ type for each IDL array. The type name is concatenated from the array name and the suffix _forany.

For example, the IDL shown earlier can be modified to include this two-dimensional array definition:

```
// IDL
typedef long longArray[2][2];
```

Given this typedef statement, the IDL compiler generates a longArray_forany type. The following example shows how to use insertion and extraction operators to move data between this type and a CORBA::Any:

```
// C++
longArray m_array = { {14, 15}, {24, 25} };

// Insertion
CORBA::Any a;
a <<= longArray_forany(m_array);

// Extraction
longArray_forany extractedValue;
if (a >>= extractedValue) {
    cout << "Element [1][2] is "
         << extractedValue[1][2] << endl;
}
```

Like array _var types, _forany types provide an operator[]() function to access array members. However, when a _forany type is destroyed the storage that is associated with the array remains intact. This is consistent with the behavior of the extraction operator >>=, where the CORBA::Any retains ownership of the memory that the operator returns. Thus, the previous code is safe from memory leaks.

Inserting and Extracting String Data

Helper types are also provided for insertion and extraction of `string` and `wstring` types.

Inserting Strings

The `from_string` and `from_wstring` struct types are used in combination with the insertion operator >>= to insert strings and wide strings. Two constructors are provided for the `from_string` type:

```
CORBA::Any::from_string(
            char* s,
            CORBA::ULong b,
            CORBA::Boolean nocopy = 0
)
CORBA::Any::from_string(const char* s, CORBA::ULong b)
```

The constructor parameters can be explained as follows:

- The `s` parameter is a pointer to the string to be inserted.
- The `b` parameter specifies the bound of a bounded string (`0` implies unbounded).
- The `nocopy` parameter specifies whether the string is copied before insertion (`0` implies copying, `1` implies no copying and adoption).

Analogous constructors are provided for the `from_wstring` type:

```
CORBA::Any::from_wstring(
            CORBA::WChar* s,
            CORBA::ULong b,
            CORBA::Boolean nocopy = 0
)
CORBA::Any::from_wstring(const CORBA::WChar* s, CORBA::ULong b)
```

Examples of inserting bounded and unbounded string types are shown in the following code:

```
// C++
// Insert a copy of an unbounded string, 'string'.
CORBA::Any a1;
a1 <<= CORBA::Any::from_string("Unbounded string", 0);
...
```

```
// Insert a copy of a bounded string, 'string<100>'.
CORBA::Any a2;
a2 <<= CORBA::Any::from_string("Bounded string", 100);
...
// Insert an unbounded string, 'string', passing
// ownership to the 'CORBA::Any'.
CORBA::Any a3;
char * unbounded = CORBA::string_dup("Unbounded string");
a3 <<= CORBA::Any::from_string(unbounded, 0, 1);
...
// Insert a bounded string, 'string<100>', passing
// ownership to the 'CORBA::Any'.
CORBA::Any a4;
char * bounded = CORBA::string_dup("Bounded string");
a3 <<= CORBA::Any::from_string(bounded, 100, 1);
```

Insertion of wide strings is performed in an analogous manner using the `CORBA::Any::from_wstring` type.

Extracting Strings

The `to_string` and `to_wstring` struct types are used in combination with the extraction operator `>>=` to extract strings and wide strings. One constructor is provided for the `to_string` type:

```
CORBA::Any::to_string(const char*& s, CORBA::ULong  b);
```

The constructor parameters can be explained as follows:

- The `s` parameter is a place holder that will point to the extracted string after a successful extraction is made.
- The `b` parameter specifies the bound of a bounded string (`0` implies unbounded).

An analogous constructor is provided for the `to_wstring` type:

```
CORBA::Any::to_wstring(const CORBA::WChar*& s, CORBA::ULong  b);
```

Examples of extracting bounded and unbounded string types are shown in the following code:

```
// C++
// Extract an unbounded string, 'string'.
CORBA::Any a1;
const char * readonly_s;
```

```
if (a1 >>= CORBA::Any::to_string(readonly_s, 0)) {
    // process string, 'readonly_s'
}
...
// Extract a bounded string, 'string<100>'.
CORBA::Any a2;
const char * readonly_bs;
if (a2 >>= CORBA::Any::to_string(readonly_bs, 100)) {
    // process bounded string, 'readonly_bs'
}
```

Extraction of wide strings is performed in an analogous manner using the `CORBA::Any::to_wstring` type.

Inserting and Extracting Alias Types

The insertion and extraction operators <<= and >>= are invalid for *alias types*. An alias type is a type defined using a `typedef`.

For example, a bounded string alias is a type defined by making a `typedef` of a bounded string:

```
//IDL
typedef string<100> BoundedString;
```

This is mapped by the IDL compiler to a C++ `typedef` as follows:

```
// C++
// Stub code generated by the IDL compiler.
typedef char* BoundedString;
...
```

A C++ alias, such as `BoundedString`, cannot be used to distinguish an overloaded operator because it is not a distinct C++ type. This is the reason why the <<= and >>= operators cannot be used with alias types.

Inserting Alias Types

The `BoundedString` alias type can be inserted into an `Any` as follows:

```
// C++
CORBA::Any a;
BoundedString bs = "Less than 100 characters.";
```

```
1 a <<= CORBA::Any::from_string(bs, 100);
2 a.type(_tc_BoundedString);  // Correct the type code!
```

The code executes as follows:

1. The data is inserted using the <<= operator and the `from_string` helper type. Initially, the Any's type code is set equal to that of a bounded string with bound `100` (the type code for `string<100>`). There is no type code constant available for the `string<100>` type—the <<= operator creates one on the fly and uses it.
2. `CORBA::Any::type()` corrects the Any's type code, setting it equal to the `_tc_BoundedString` type code.

   It is not permissible to use `type()` to reset the type code to arbitrary values—the new type code must be equivalent to the old one. Attempting to reset the type code to a non-equivalent value raises the `BAD_TYPECODE` system exception.

   For example, calling `type()` with the `_tc_BoundedString` argument succeeds because the `BoundedString` type is equivalent to the `string<100>` type.

Extracting Alias Types

The `BoundedString` alias type can be extracted from an `Any` as follows:

```
// C++
CORBA::Any a;
// The any 'a' is initialized with a 'BoundedString' alias
// (as shown previously)
...
// Extract the 'BoundedString' type
1 const char * bs;
2 if (a >>= CORBA::Any::to_string(bs, 100) ) {
      cout << "Bounded string is: \"" << bs << "\"" << endl;
  }
```

1. The pointer to receive the extracted value, `bs`, is declared as `const char*`. You cannot declare `bs` as `const BoundedString` because that means a const pointer to char, or `char* const` which is not the same as `const char*` (pointer to const char).

2. The `to_string` constructor manufactures a type code for a `string<100>` bounded string and compares this type with the Any's type code. If the type codes are equivalent, the extraction succeeds.

Querying a CORBA::Any's Type Code

Type code operations are commonly used to query a `CORBA::Any` for its type. For example, given this interface definition:

```
// IDL
struct Example {
    long l;
};
```

the IDL compiler generates the `CORBA::TypeCode_ptr` constant `_tc_Example`.

Assume that a client program invokes the IDL operation `op()`:

```
// IDL
interface Bar {
    void op(in any a);
};
```

as follows:

```
// C++
// Client code
Bar_var bVar;
CORBA::Any a = ... ; // somehow initialize
...
bVar->op(a);
```

The server can query the actual type of the parameter to `op()` as follows:

```
// C++
// Server code
void Bar_i::op(const CORBA::Any& a) {
    CORBA::TypeCode_var t(a->type());
    if(t->equivalent(_tc_Example)) {
        cerr << "Don't like struct Example!" << endl;
    }
    else...   // Continue processing here.
}
```

This is one of the most common uses of TypeCodes—namely, the runtime querying of type information from a CORBA::Any.

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 28.

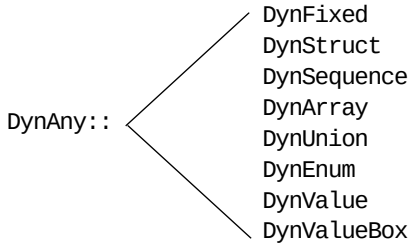

Figure 28: *Interfaces that derive from the DynAny interface*

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);
  };
};
```

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).

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.

Using 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:

```
//C++
#include <omg/DynamicAny.hh>
//...
void get_any_val(const CORBA::Any& a){
1     // Get a reference to a 'DynamicAny::DynAnyFactory' object
      CORBA::Object_var obj
          = global_orb->resolve_initial_references("DynAnyFactory");
      DynamicAny::DynAnyFactory_var dyn_fact
          = DynamicAny::DynAnyFactory::_narrow(obj);
      if (CORBA::is_nil(dyn_fact)) {
          // error: throw exception
      }

      // Get the Any's type code
      CORBA::TypeCode_var tc = a.type();
2     switch (tc->kind()){
      // ...
        case CORBA::tk_enum: {
3           DynamicAny::DynAny_var da = dyn_fact->create_dyn_any(a);
            DynamicAny::DynEnum_var de =
                DynamicAny::DynEnum::_narrow(da);
            // ...
4           de->destroy();
        }
        break;
      }
}
```

The code executes as follows:

1. Call `resolve_initial_references("DynAnyFactory")` to obtain an initial reference to the `DynAnyFactory` object.

   It is assumed that `global_orb` refers to an existing `CORBA::ORB` object that has been initialized prior to this code fragment.

   Narrow the `CORBA::Object_ptr` object reference to the `DynamicAny::DynAnyFactory_ptr` type before it is used.
2. Analysis of a type code is begun by branching according to the value of its kind field. A general purpose subroutine for processing `DynAnys` would require case statements for every possible IDL construct. Only the case statement for an `enum` is shown here.
3. 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 `DynamicAny::DynEnum_ptr`. The return value can therefore be narrowed to this type.
4. `destroy()` must be invoked on the `DynAny` object when you are finished with it.

Using 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 C++ you can insert this anonymous bounded string using the `CORBA::Any::from_string` helper type. Alternatively, you can use the `DynamicAny` programming interface as follows:

```
//C++
#include <omg/DynamicAny.hh>
//...
// Get a reference to a 'DynamicAny::DynAnyFactory' object
CORBA::Object_var obj
    = global_orb->resolve_initial_references("DynAnyFactory");
DynamicAny::DynAnyFactory_var dyn_fact
    = DynamicAny::DynAnyFactory::_narrow(obj);
if (CORBA::is_nil(dyn_fact)) {
    // error: throw exception
}
```

```
   // Create type code for an anonymous bounded string type
   CORBA::ULong bound = 128;
2  CORBA::TypeCode_var tc_v = global_orb->create_string_tc(bound);

   // Initialize a 'DynAny' containing a bounded string
3  DynamicAny::DynAny_var 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  CORBA::Any_var a = dyn_bounded_str->to_any();
   //...
   // Cleanup 'DynAny'
6  dyn_bounded_str->destroy();
```

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")`.

   It is assumed that `global_orb` refers to an existing `CORBA::ORB` object that has been initialized prior to this code fragment.

   The plain `CORBA::Object_ptr` object reference must be narrowed to the `DynamicAny::DynAnyFactory_ptr` type before it is used.
2. The `CORBA::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()`. The `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 equal to the given argument of the `insert_string()` operation. Insertion operations, of the form `insert_`*`BasicType`*, are defined for all basic types as described in "Accessing Basic DynAny Values" on page 322.
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 327.
- 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 327.

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 304 and see page 306). 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 329).

The IDL for insertion and extraction operations is shown in the following sections.

Insertion Operations

The IDL insertion operations supported by the DynAny interface are:

```
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:

```
//C++
#include <omg/DynamicAny.hh>
//...
// Get a reference to a 'DynamicAny::DynAnyFactory' object
CORBA::Object_var obj
    = global_orb->resolve_initial_references("DynAnyFactory");
DynamicAny::DynAnyFactory_var dyn_fact
    = DynamicAny::DynAnyFactory::_narrow(obj);
if (CORBA::is_nil(dyn_fact)) {
    // error: throw exception
}

// create DynAny with a string value
DynamicAny::DynAny_var dyn_a;
dyn_a = dyn_fact->create_dyn_any_from_type_code(
                      CORBA::_tc_string
        );
dyn_a->insert_string("not to worry!");

// convert DynAny to any
CORBA::Any_var a;
a = dyn_a->to_any();
//...
// destroy the DynAny
dyn_a->destroy();
```

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:

```
//C++
#include <omg/DynamicAny.hh>
//...
// Get a reference to a 'DynamicAny::DynAnyFactory' object
CORBA::Object_var obj
    = global_orb->resolve_initial_references("DynAnyFactory");
DynamicAny::DynAnyFactory_var dyn_fact
    = DynamicAny::DynAnyFactory::_narrow(obj);
if (CORBA::is_nil(dyn_fact)) {
    // error: throw exception
}

CORBA::Any a = ...; // get Any from somewhere

// create DynAny from Any
DynamicAny::DynAny_var dyn_a = dyn_fact->create_dyn_any(a);

// get DynAny's type code
CORBA::TypeCode_var tcode = dyn_a->type();

// evaluate type code
switch(tcode->kind()){
case CORBA::tk_short:
    {
        CORBA::Short s = dyn_a->get_short();
        cout << "any contains short value of " << s << endl;
        break;
    }
case CORBA::tk_long:
    {
        CORBA::Long l = dyn_a->get_long();
        cout << "any contains long value of " << l << endl;
        break;
    }
// other cases follow
...
} // end of switch statement

dyn_a->destroy(); // cleanup
```

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
- DynArray
- DynFixed
- DynValue
- DynValueBox

DynEnum

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

- 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().
- Operations 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:

```
//C++
void extract_any(const CORBA::Any * a){
    //...
    // Get a reference to a 'DynamicAny::DynAnyFactory' object
    CORBA::Object_var obj
        = global_orb->resolve_initial_references("DynAnyFactory");
    DynamicAny::DynAnyFactory_var dyn_fact
        = DynamicAny::DynAnyFactory::_narrow(obj);
    if (CORBA::is_nil(dyn_fact)) {
        // error: throw exception
    }

    DynamicAny::DynAny_var dyn_a = dyn_fact->create_dyn_any(*a);
    CORBA::TypeCode_var tcode = dyn_a->type();

    switch(tcode->kind()){
        case CORBA::tk_enum:
        {
            DynamicAny::DynEnum_var dyn_e =
                DynamicAny::DynEnum::_narrow(dyn_a);
            CORBA::String_var s = dyn_e->get_as_string();
            cout << s << endl;
            dyn_e->destroy();
        }
        // other cases follow
        // ...
    }
}
```

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 the eval_member() helper function for further decomposition:

```
//C++
DynamicAny::DynStruct_var dyn_s = ...;
CORBA::TypeCode_var tcode = dyn_s->type();
int counter = tcode->member_count();

for (int i = 0; i < counter; i++) {
    DynamicAny::DynAny_var 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 327. For example, the following code iterates over elements in a `DynArray`:

```
DynamicAny::DynArray_var dyn_array = ...;
CORBA::TypeCode_var tcode = dyn_array->type();
int counter = tcode->length();

for (int i = 0; i < counter; i++){
    DynamicAny::DynAny_var 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`.

16 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.

Using the DII

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.

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, this 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, Orbix raises an exception to the gateway, which in turn can report an error to the caller.

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.

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)
        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:

```
bankVar->newAccount(ownerName, initialBalance);
```

Constructing a Request Object

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.

Using _request()

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.

Create a Request Object

Call `_request()` on the target object and specify the name of the operation to invoke:

```
// Get object reference
CORBA::Object_var target = ... ;
```

```
// Create Request object for operation newAccount()
CORBA::Request_var 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 TypeCode constants, refer to "Type Code Constants" on page 301.

For information about supported TypeCodes, see Chapter 14 on page 293.

Set Operation Parameters

A request object uses an NVList to store the data for an operation's parameters. You set the NVList by either specifying each parameter, or reading an operation definition from the interface repository. For information on both methods, see "Setting Request Object Parameters" on page 343.

Using _create_request()

You can create a Request object by calling _create_request() on an object reference and passing the request details as arguments. At a minimum, you must provide two arguments:

- The name of the operation
- A pointer to a NamedValue that holds the operation's return value

You can also supply an NVList that is already populated with the operation's parameter data. If you supply null for the NVList argument, _create_request() creates an empty NVList for the returned Request object.

In either case, you set the NVList's parameters one at a time, or by reading an operation definition from the interface repository. For information on both methods, see "Setting Request Object Parameters".

For example, the following code constructs a Request object for invoking operation newAccount():

```
// get an object reference
CORBA::Object_var target = ... ;

CORBA::Request_ptr newAcctRequest;
CORBA::NamedValue_ptr result;

// Construct the Request object
target->_create_request(
    CORBA::Context::_nil(), "newAccount", CORBA::NVList::_nil(),
    result, newAcctRequest,  0 );
```

Setting Request Object Parameters

A request object uses an NVList to store the data for an operation's parameters, where each NVList element—a NamedValue object—holds the data for a single parameter—its direction (in, out, or inout) and the argument that it passes.

You can set an operation's parameters in one of two ways:

- Add each parameter individually.
- Build the NVList from the interface repository.

Adding Parameters

You add each parameter to a Request object's NVList in one of two ways:

- Invoke arguments() on the Request object to obtain its NVList; then populate the NVList with one of its methods:

  ```
  add()
  add_item()
  add_item_consume()
  add_value()
  add_value_consume()
  ```

- Use one of several shortcut methods provided by the Request object that let you populate the Request object's NVList with the desired parameter information. Request objects contain methods for each direction type:

  ```
  add_in_arg();
  add_inout_arg();
  add_out_arg();
  ```

For example, you can populate the empty NVList of request object newAcctRequest as follows:

```
// C++
req->add_in_arg() <<= "Chris";
CORBA::NamedValue_ptr 1000.00;
```

Note: You can use these shortcut methods only if you create a Request object whose NVList is initially empty.

Setting Parameters From the Interface Repository

A client can use an operation definition in the interface repository to build a Request object's NVList. The interface repository describes operations through CORBA::OperationDef objects.You can read an operation's parameters into a Request object's empty NVList as follows:

1. Call arguments() on the request object to get a pointer to its NVList.
2. Call create_operation_list() on the ORB and supply it a reference to the desired OperationDef object and the empty NVList.

   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.
3. Supply arguments to the operation parameters by iterating over the NVList elements with NVList::item(). Use the insertion operator <<= to set each NamedValue's value member.

Invoking a Request

After you set a Request object's data, you can use one of several methods 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 346).

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:

```
// C++
try {
    if (request->invoke())
        // Call to invoke() succeeded
    else
        // Call to invoke() failed.
}
catch (CORBA::SystemException& se) {
    cout << "Unexpected exception" << &se << endl;
}
```

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 pointer 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` pointer.
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:

```
CORBA::Object_var newAccount;
request->return_value() >>= newAccount;
// narrow account object ...
```

Getting Information about a Request Object

Given a Request object, you can get its operation name and target object reference by calling operation() and target() on it, respectively.

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.

You can also invoke methods asynchronously. For more information, see Chapter 12.

Using the DSI

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.

DSI Applications

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.

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.

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

The DSI is implemented by servants that instantiate dynamic skeleton classes. All dynamic skeleton classes are derived from `PortableServer::DynamicImplementation`:

```
namespace Portable Server{
    class DynamicImplementation : public virtual ServantBase{
        public:
            Object_ptr _this();
            virtual void invoke( ServerRequest_ptr request ) = 0;
            virtual RepositoryId _primary interface(
                const ObjectId& oid, POA_ptr poa) = 0;
    };
}
```

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.

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 C++ class:

```
class ServerRequest{
    public:
        const char* operation() cont;
        void arguments( NVList_ptr& parameters);
        Context_ptr ctx();
        void set_result(const Any& value);
        void set_exception(const Any& value);
};
```

`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.

17 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.

These facilities 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
```

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 22, 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 363.

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 361.

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 356.

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 22:

Table 22: *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 23 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 23 on page 360 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 23 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 23 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.
`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.

Table 22: *Interface Repository OIbject Types*

Object type	Description
`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.
`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 29 shows the hierarchy for all interface repository objects.

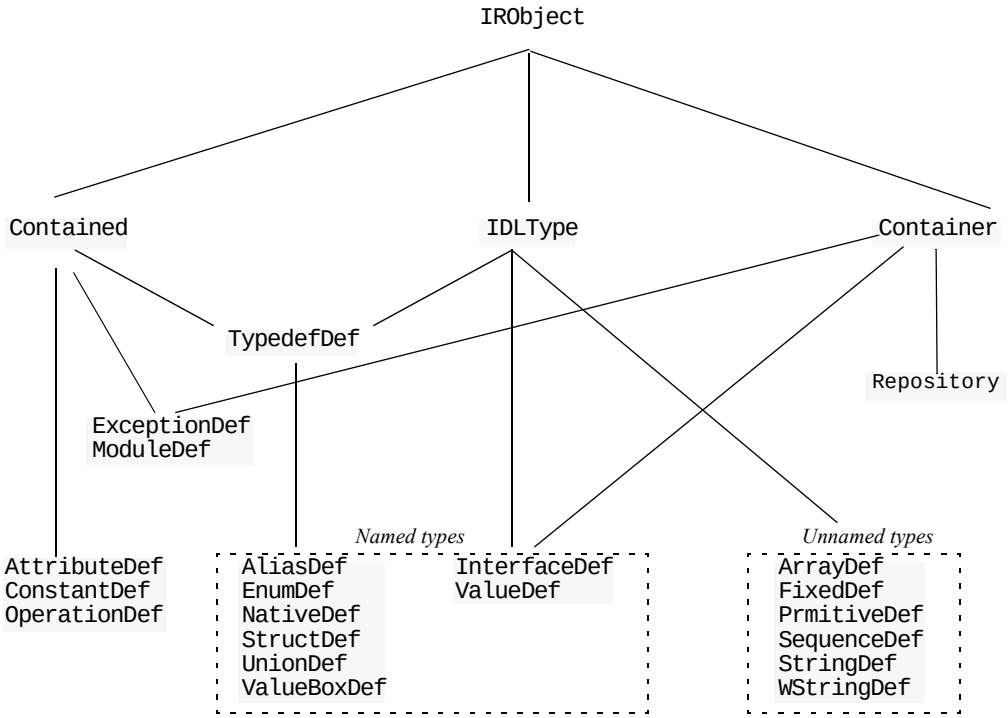

Figure 29: *Hierarchy of interface repository objects*

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 and 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 365.

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.

The interface repository abstracts the properties of containment into two abstract base interfaces, `Container` and `Contained`. 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`.

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`.

Table 23 shows the relationship between `Container` and `Contained` objects in the interface repository.

Table 23: *Container and Contained Objects in the Interface Repository*

Container object type	Contained Objects

`Repository`	`ConstantDef`
	`TypedefDef`
	`ExceptionDef`
	`InterfaceDef*`
	`ModuleDef*`
	`ValueDef*`

Table 23: *Container and Contained Objects in the Interface Repository*

Container object type	Contained Objects

`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
    );
};
```

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`.

`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;
};
```

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 365).

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
};
```

The container interface provides four lookup functions 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 368.

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.

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;
};
```

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`). 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 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.

Several repository object types use the `TypeDescription` structure to store their information: `EnumDef`, `UnionDef`, `AliasDef`, and `StructDef`.

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:

```
// C++
CORBA::InterfaceDef_var 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

`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:

```
CORBA::Contained_var 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:

```
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::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 365).

Container::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.

```
// C++
int i;
Repository_var rVar;
Contained_var cVar;
InterfaceDef_var interfaceVar;
InterfaceDef::FullInterfaceDescription_var full;
CORBA::Object_var obj;

try {
```

```
    // get an object reference to the IFR:
    obj = orb->resolve_initial_references("InterfaceRepository");
    rVar = Repository::_narrow(obj);

    // Get the interface definition:
    cVar = rVar->lookup("grid");
    interfaceVar = InterfaceDef::_narrow(cVar);

    // Get a full interface description:
    full = interfaceVar->describe_interface();
    // Now print out the operation names:
    cout << "The operation names are:" << endl;
    for (i=0; i < full->operations.length(); i++)
        cout << full->operations[i].name << endl;
    // Now print out the attribute names:
    cout << "The attribute names are:" << endl;
    for (i=0; i < full->attributes.length(); i++)
        cout << full->attributes[i].name << endl;
}
catch (...) {
    ...
}
```

The example can be extended by finding the `OperationDef` object for an operation called `doit()`. Operation `Container::lookup_name()` can be used as follows:

```
// C++
ContainedSeq_var opSeq;
OperationDef_var doitOpVar;

try {
    cout << "Looking up operation doit()"
            << endl;
    opSeq = interfaceVar->lookup_name(
                    "doit", 1, dk_Operation, 0);
    if (opSeq->length() != 1) {
        cout << "Incorrect result for lookup_name()";
        exit(1);
```

```
    } else {
        // Narrow the result to be an OperationDef.
        doitOpVar =
                OperationDef::_narrow(opSeq[0]))
    }
    ...
}
catch (...) {
    ...
}
```

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 Format

This is the default format that Orbix uses. It is derived from the IDL definition's scoped name and contains three colon-delimited components as follows:

`IDL:`*`identifier[/identifier]...`*`:`*`version-number`*

- 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 Format

The DCE UUID has the following format:

`DCE:`*`UUID`*`:`*`minor-version-number`*

LOCAL Format

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 372).

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.

18 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. 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.

This chapter describes how to build and maintain naming graphs programmatically. It also shows how to use object groups to achieve load balancing. 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 *Orbix 2000 Administrator's Guide*.

Overview

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*.

Figure 30 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.

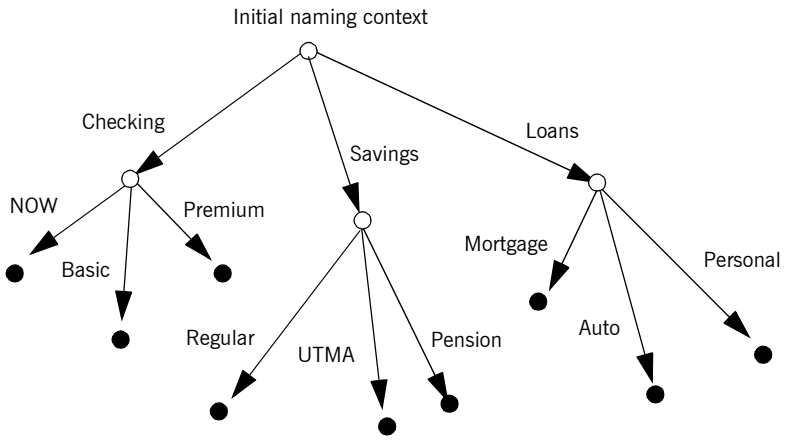

Figure 30: *A naming graph is a hierarchy of naming contexts*

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

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.

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 30 on page 378, the path to a Personal loan account object is specified by a `Name` sequence in which only the `id` fields are set:

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:

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 381).
- `to_string()` converts a `Name` to a `StringName` (see page 382).
- `resolve_str()` uses a `StringName` to find a `Name` in a naming graph and returns an object reference (see page 389).

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:

```
CosNaming::Name name(2);
name.length(2);
name[0].id = CORBA::string_dup("Loans");
name[0].kind = CORBA::string_dup( "" );
name[1].id = CORBA::string_dup("Personal");
name[1].kind = CORBA::string_dup( "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 382):

```
// get initial naming context
CosNaming::NamingContextExt_var root_cxt = ...;

CosNaming::Name_var name;
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:

Index	id	kind
0	Loans	
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`:

```
// get initial naming context
CosNaming::NamingContextExt_var root_cxt = ...;
CosNaming::NamingContextExt::StringName str_n;

// initialize name
CosNaming::Name_var name = ...;
...
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:

```
...
// Initialize the ORB
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Get reference to initial naming context
CORBA::Object obj =
    orb_var->resolve_initial_references("NameService");
```

To obtain a reference to the naming context, narrow the result with `CosNaming::NamingContextExt::_narrow()`:

```
CosNaming::NamingContextExt_var root_cxt;
if (root_cxt =
        CosNaming::NamingContextExt::_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 382).
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 30 on page 378 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 31:

```
//get initial naming context
CosNaming::NamingContextExt_var root_cxt = ...;

CosNaming::NamingContext_var checking_cxt;

// create naming context
checking_cxt = root_cxt->new_context();

// initialize name
CosNaming::Name_var name;
name.length(1);
name[0].id = CORBA::string_dup("Checking");
name[0].kind = CORBA::string_dup( "" );

// bind new context
root_cxt->bind_context(name, checking_cxt);
```

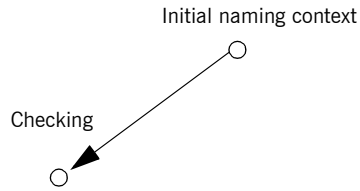

Figure 31: *Checking context bound to initial naming context*

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.

```
CosNaming::NamingContext_var 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);
```

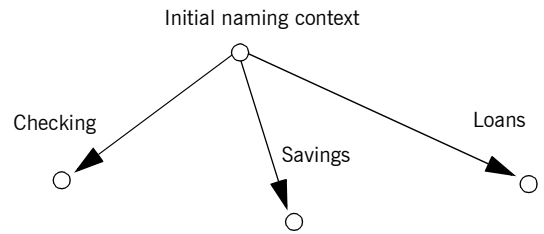

Figure 32: *Savings and Loans naming contexts bound to initial naming context*

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 396).

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 388). The following code shows how you might orphan the Savings naming context:

```
//get initial naming context
CosNaming::NamingContextExt_var root_cxt = ...;

CosNaming::NamingContext_var savings_cxt;

// initialize name
CosNaming::Name_var name;
name.length(1);
name[0].id = CORBA::string_dup("Savings");
name[0].kind = CORBA::string_dup( "" );

// create and bind checking_cxt
savings_cxt = root_cxt->bind_new_context(name);

// make another context
CosNaming::NamingContext_var 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 395).

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:

```
// object reference "basic_check" obtained earlier
...

name->length(1);
name[0].id = CORBA::string_dup("Basic");
name[0].kind = CORBA::string_dup("");
checking_cxt->bind(name, basic_check);
```

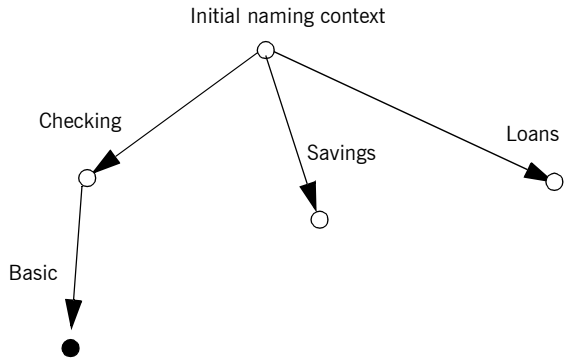

Figure 33: *Binding an object reference to a naming context*

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 386). 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 382).

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:

```
CosNaming::Name_var name;
name.length(2);
name[0].id = CORBA::string_dup("Checking");
name[0].kind = CORBA::string_dup("");
name[1].id = CORBA::string_dup("Basic");
name[1].kind = CORBA::string_dup("");
```

- If the client code narrows the initial naming context to the NamingContextExt interface, it can call to_name() on the initial naming context. This operation takes a CosNaming::CosNamingExt::StringName argument and returns a CosNaming::Name as follows:

```
CosNaming::Name_var name;
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 381.

Resolving Names

Clients call resolve() on the initial naming context to obtain the object associated with the supplied name:

```
CORBA::Object_var 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:

```
CORBA::Object_var 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_var checking_var;
...
try {
    checking_var = BasicChecking::_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 162). 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:

```
CORBA::Object_var 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 380). 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
```

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 30, a client application can invoke `list()` on the initial naming context and return a binding list with three `Binding` elements:

Index	Name	BindingType
0	Checking	ncontext
1	Savings	ncontext
2	Loan	ncontext

Using a Binding Iterator

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();
}
```

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:

```
// printing function
void
print_binding_list(const CosNaming::BindingList &bl)
{
    for( CORBA::Ulong i = 0; i < bl.length(); i++ ){
        cout << bl[i].binding_name[0].id;
        if( bl[i].binding_name[0].kind != '\0' )
            cout << "(" << bl[i].binding_name[0].kind << ")";
        if( bl[i].binding_type == CosNaming::ncontext )
            cout << ": naming context" << endl;
```

```
        else
            cout << ": object reference" << endl;
    }
}

void
get_context_bindings(CosNaming::NamingContext_ptr cxt)
{
    CosNaming::BindingList_var b_list;
    CosNaming::BindingIterator_var b_iter;
    const CORBA::ULong MAX_BINDINGS = 50;

    if (!CORBA::is_nil(cxt)) {

        // get first set of bindings from cxt
        root_cxt->list(MAX_BINDINGS, b_list, b_iter);

        //print first set of bindings
        print_binding_list(b_list);

        // look for remaining bindings
        if( !CORBA::is_nil(b_iter) ) {
            CORBA::Boolean more;
            do {
                is_nil(b_iter) ) {
                more = b_iter->next_n(MAX_BINDINGS, b_list);
                // print next set of bindings
                print_binding_list(b_list);
            } while (more);
        }
        // get rid of iterator
        b_iter->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 386). 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 34, you can destroy the Loans context and its bindings to the loan account objects as follows:

```
CosNaming::Name_var name;

// get initial naming context
CosNaming::NamingContextExt_var root_cxt = ...;

// assume availability of Loans naming context variable
CosNaming::NamingContext_var 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);

// destroy orphaned Loans context
loans_cxt->destroy();
```

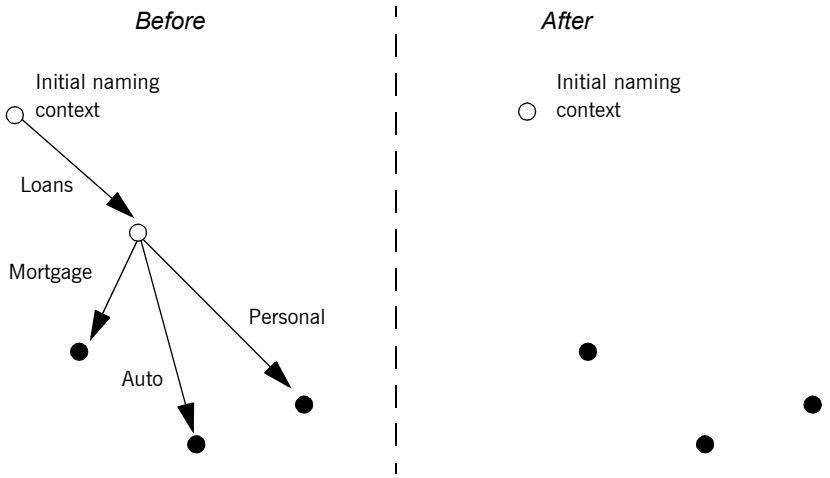

Figure 34: *Destroying a naming context and removing related bindings*

Note: Orbix provides administrative tools to destroy contexts and remove bindings. These are described in the *Orbix 2000 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*.

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 Structures

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:

- 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.
- Hierarchical 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.

Figure 35 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.

In this hierarchical structure, the naming graphs in the Deposits and Loans

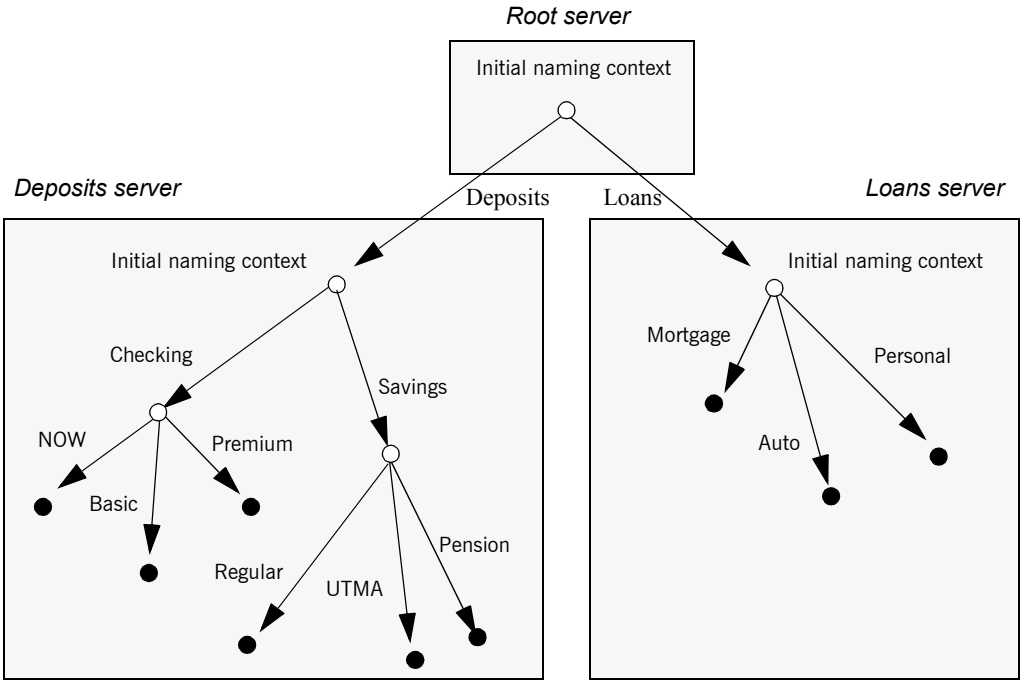

Figure 35: *A naming graph that spans multiple servers*

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:

```
// Root server
#include <omg/CosNaming.hh>
...
int main (int argc, char** argv) {
    CosNaming::NamingContextExt_var
        root_inc, deposits_inc, loans,_inc;
```

```
    CosNaming::Name_var name;
    CORBA::Object_var obj;
    CORBA::ORB_var orb_var;
    char *loans_inc_ior, deposits_inc_ior
...
    try {
        orb_var = CORBA::ORB_init(argc, argv, "Orbix");

        // code to obtain stringified IORs of initial naming
        // contexts for Loans and Deposits servers (not shown)
        ...

        obj = orb_var->string_to_object (loans_inc_ior);
        loans_inc ==
            CosNaming::NamingContextExt::_narrow(obj);
        obj = orb_var->string_to_object (deposits_inc_ior);
        deposits_inc ==
            CosNaming::NamingContextExt::_narrow(obj);

        // get initial naming context for Root server
        root_inc = ... ;

        // bind Deposits initial naming context to root server
        // initial naming context
        name = root_inc->to_name("Deposits");
        root_inc->bind_context(name, deposits_inc);

        // bind Loans initial naming context to root server's
        // initial naming context
        name = root_inc->to_name("Loans");
        root_inc->bind_context(name, deposits_inc);
    }
}
```

This yields the following bindings between the three naming graphs:

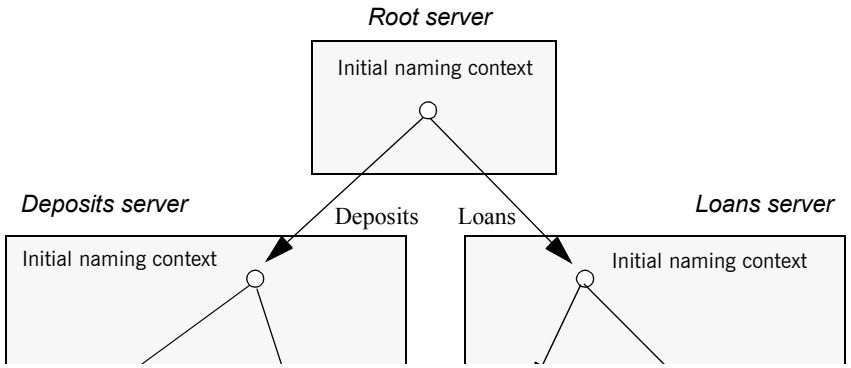

Figure 36: *Multiple naming graphs are linked by binding initial naming contexts of several servers to a root server.*

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 37 shows how this federates the three naming graphs:

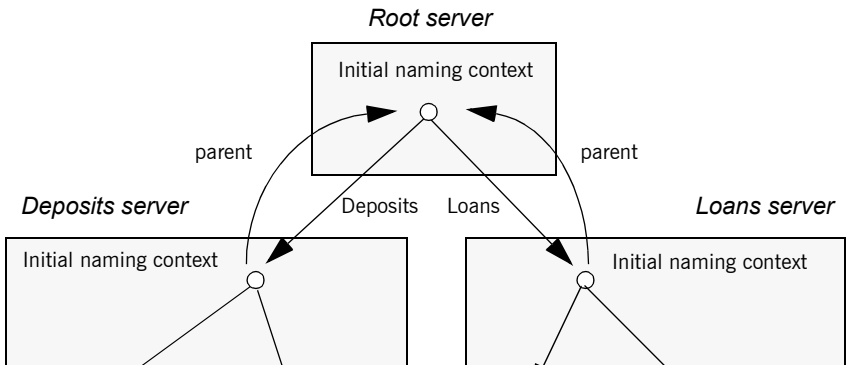

Figure 37: *The root server's initial naming context is bound to the initial naming contexts of other servers, allowing clients to locate the root naming context.*

The code for both Deposits and Loans server processes is virtually identical:

```
#include <omg/CosNaming.hh>
...
int main (int argc, char** argv) {
    CosNaming::NamingContextExt_var
        root_inc, this_inc;
    CosNaming::Name_var name;
    CORBA::Object_var obj;
    CORBA::ORB_var orb_var;
    char *root_inc_ior;
    ...
    try {
        orb_var = CORBA::ORB_init (argc, argv, "Orbix");

        // code to obtain stringified IORs of root server's
        // initial naming context (not shown)
        ...

        obj = orb_var->string_to_object (root_inc_ior);
        root_inc ==
            CosNaming::NamingContextExt::_narrow(obj);
```

```
        // get initial naming context for this server
        this_inc = ... ;

        name = this_inc->to_name("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

```
// C++
#include <omg/CosNaming.hh>
...
int main (int argc, char** argv) {
    CosNaming::NamingContextExt_var root_cxt;
    CosNaming::NamingContext_var
        checking_cxt, savings_cxt, loan_cxt;
    CosNaming::Name_var name;
    CORBA::ORB_var orb;
    CORBA::Object_var obj;
    Checking_var basic_check, now_check, premium_check;
    // Checking_var objects initialized from
    // persistent data (not shown)

    try {
        // Initialize the ORB
        orb = CORBA::ORB_init(argc, argv, "Orbix");

        // Get reference to initial naming context
        obj =
            orb_var->resolve_initial_references("NameService");
        root_cxt = CosNaming::NamingContextExt::_narrow(obj))
```

```
        if( !CORBA::is_nil(root_cxt) ){
            // 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);

            // 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

```
// C++
#include <omg/CosNaming.hh>
...
int main (int argc, char** argv) {
    CosNaming::NamingContextExt_var root_cxt;
    CosNaming::Name_var name;
```

```
    BasicChecking_var checking_var;
    CORBA::Object_var obj;
    CORBA::ORB_var orb_var;

    try {
        orb_var = CORBA::ORB_init (argc, argv, "Orbix");

        // Find the initial naming context
        obj =
            orb_var->resolve_initial_references("NameService");
        if (root_cxt ==
            CosNaming::NamingContextExt::_narrow(obj)) {
            obj = root_cxt->resolve_str("Checking/Basic");
            if (checking_var == BasicChecking::_narrow(obj)) {
                // perform some operation on basic checking object
                ...
            }
            else { ... } // Deal with failure to _narrow()
        } else { ... } // Deal with failure to _narrow()

    } // 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.

Each object group has a selection algorithm, which is set on the object group's creation (see page 408). This algorithm is applied when a client resolves the name associated with the object group. Three algorithms are supported:

- Round-robin selection
- Random selection
- Active load balancing selection

Object groups provide a way to replicate frequently requested objects, and thereby distribute the request processing load. The naming service directs client requests to the various replicated objects according to the object group's selection algorithm. The existence of an object group is transparent to the client, which resolves the object group name in the same way that it resolves any other name.

If you choose the active load balancing algorithm, each object in an object group is assigned a load count that is periodically updated by servers. The naming service directs client requests to the object with the lowest load count.

Figure 38 shows how a name can bind to multiple objects through an object group.

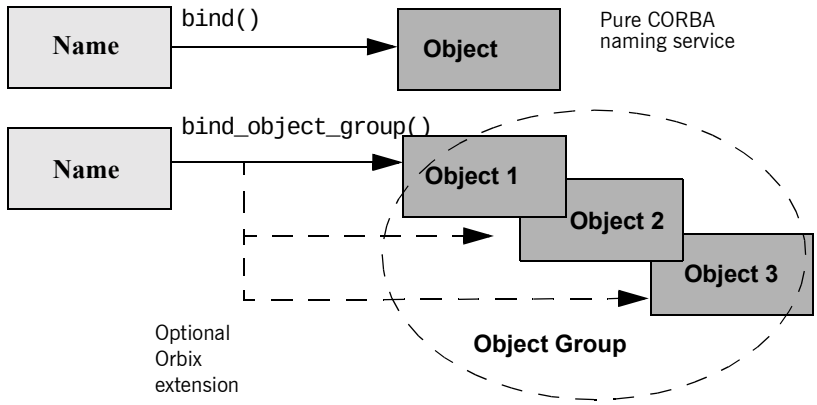

Figure 38: *Associating a name with an object group*

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 *Orbix 2000 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.

Creating 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.

Adding 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.

Removing 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.

Removing 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.

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 39 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.

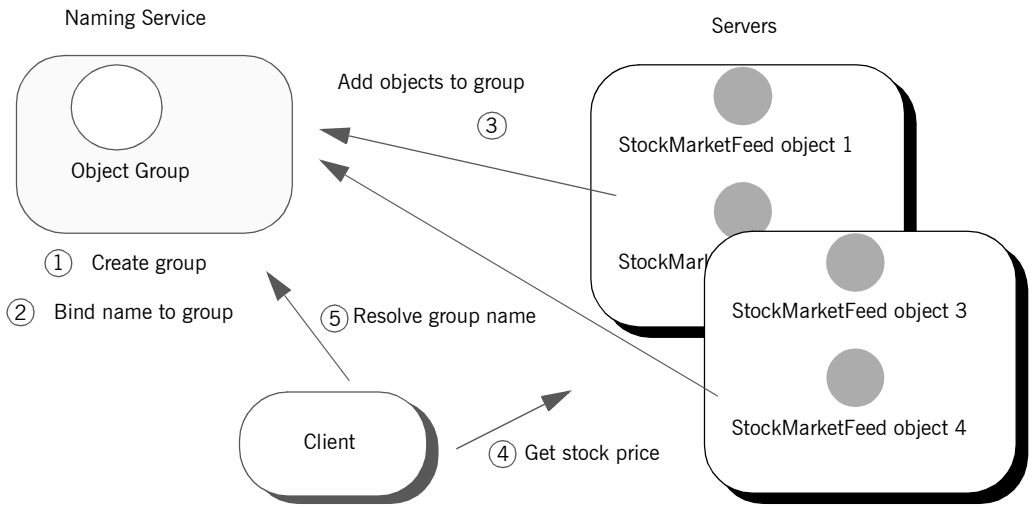

Figure 39: *Architecture of the stock market example*

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:

```
#include <stdlib.h>
#include <string.h>
#include <iostream.h>
#include <omg/orb.hh>
#include <omg/PortableServer.hh>
#include <it_ts/termination_handler.h>
```

```
#include <orbix/naming.hh>
#include "stock_i.h"

static CORBA::ORB_var global_orb = CORBA::ORB::_nil();
static PortableServer::POA_var the_poa;

// Needed in global scope so it's available to
   termination_handler():
IT_LoadBalancing::ObjectGroup_var rr_og_var;
IT_Naming::IT_NamingContextExt_var it_ins_var;
CosNaming::Name_var nm;
char id1[100], id2[100];

static void
termination_handler(long sig)
{
    try
    {
        cout << "Removing members: " << id1 << " and "
             << id2 << endl;
        rr_og_var->remove_member(id1);
        rr_og_var->remove_member(id2);
    }
    catch (...)
    {
        cerr << "Could not remove members." << endl;
    }
    IT_LoadBalancing::MemberIdList_var members =
        rr_og_var->members();
    if (members->length() == 0)  // Last one to remove members
    {
        try
        {
            cout << "Unbinding object group..." << endl;
            it_ins_var->unbind(nm);
            cout << "Destroying group..." << endl;
            rr_og_var->destroy();
        }
        catch (...)
        {
            cerr << "Unbind/destroy failed." << endl;
        }
    }
```

```
    cout << "Shutting down the ORB." << endl;
    global_orb->shutdown(0);
}


int
main(
    int argc,
    char *argv[]
)
{
    if (argc != 2)
    {
        cerr << "Usage: ./server <name>" << endl;
        return 1;
    }

    CORBA::String_var server_name = CORBA::string_dup(argv[1]);

    try
    {
        global_orb = CORBA::ORB_init(argc, argv);
    }
    catch (CORBA::Exception &ex)
    {
        cerr << "Could not initialize the ORB." << endl;
        cerr << "Exception info: " << ex << endl;
        return 1;
    }
    IT_TerminationHandler::set_signal_handler(
        termination_handler);

    // Initialize the POA and POA Manager:
    //
    PortableServer::POAManager_var poa_manager;
    try
    {
        CORBA::Object_var poa_obj =
            global_orb->resolve_initial_references("RootPOA");
        the_poa = PortableServer::POA::_narrow(poa_obj);
        poa_manager = the_poa->the_POAManager();
    }
    catch (CORBA::Exception &ex)
```

```
        {
            cerr << "Could not obtain the RootPOA or the POAManager."
                 << endl;
            cerr << "Exception info: " << ex << endl;
            return 1;
        }

1       // Create 2 StockMarketFeed objects <server_name>:RR_Member1
        // and<server_name>:RR_Member2.
        strcpy(id1,server_name.in());
        strcat(id1,":");
        strcat(id1,"RR_Member1");
        strcpy(id2,server_name.in());
        strcat(id2,":");
        strcat(id2,"RR_Member2");
        StockServantFeedServant *stk_svnt1 =
            new StockServantFeedServant(id1);
        StockServantFeedServant *stk_svnt2 =
            new StockServantFeedServant(id2);

2       // Resolve naming service and narrow to the interface with IONA
        // load balancing extensions, and get the object group factory
        //
        CORBA::Object_var ins_obj;
        IT_LoadBalancing::ObjectGroupFactory_var ogf_var;
        try
        {
            ins_obj =
                global_orb->resolve_initial_references("NameService");
            it_ins_var =
                IT_Naming::IT_NamingContextExt::_narrow(ins_obj);
3           ogf_var = it_ins_var->og_factory();
        }
        catch (CORBA::Exception &ex)
        {
            cerr << "Could not obtain or _narrow() reference to "
                 << "IT_Naming::IT_NamingContextExt " << endl
                << "interface. Is the Naming Service running?" << endl;
            cerr << "Exception info: " << ex << endl;
            return 1;
        }
```

```
        // Create a round robin object group and bind it in the
        // naming service
        CORBA::String_var rr_id_str =
            CORBA::string_dup("StockFeedGroup");
        try
        {
4           rr_og_var = ogf_var->create_round_robin(rr_id_str);
            nm = it_ins_var->to_name("StockSvc");
5           it_ins_var->bind_object_group(nm,rr_og_var);
        }
        catch (...)
        {
            // OK: assume other server created object group and
            // bound it in NS
            rr_og_var = ogf_var->find_group(rr_id_str);
        }

        // Add the StockMarketFeed objects to the Object Group:
6       try
        {
            IT_LoadBalancing::Member member_info;

            member_info.id = CORBA::string_dup(id1);
            member_info.obj = stk_svnt1->_this();
            rr_og_var->add_member(member_info);

            member_info.id = CORBA::string_dup(id2);
            member_info.obj = stk_svnt2->_this();
            rr_og_var->add_member(member_info);
        }
        catch (CORBA::Exception &ex)
        {
            cerr << "Could not add members " << id1 << " , "
                 << id2 << endl;
            cerr << "Exception info: " << ex << endl;
            return 1;
        }

        // Start accepting requests
        try
        {
            poa_manager->activate();
```

```
            cout << "Server ready..." << endl;
7           global_orb->run();
        }
        catch (CORBA::Exception &ex)
        {
            cerr << "Could not activate the POAManager,
                    or orb->run() failed."
                 << endl;
            cerr << "Exception info: " << ex << endl;
            return 1;
        }

        return 0;
    }
```

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:

```
#include <iostream.h>
#include <omg/orb.hh>
#include <orbix/naming.hh>
#include "stock_demo.hh"

static CORBA::ORB_var global_orb = CORBA::ORB::_nil();

int
main(
    int argc,
    char *argv[]
)
{
    if (argc != 2) {
        cerr << "Usage: ./client <stock_symbol>" << endl;
        return 1;
    }

    CosNaming::NamingContextExt_var ins;

    try {
        global_orb = CORBA::ORB_init(argc, argv);
```

```
        CORBA::Object_var ins_obj =
            global_orb->resolve_initial_references("NameService");
        ins = CosNaming::NamingContextExt::_narrow(ins_obj);
    }
    catch (CORBA::Exception &ex){
        cerr << "Cannot resolve/narrow the NameService IOR."
             << endl;
        cerr << "Exception info: " << ex << endl;
        return 1;
    }

    StockDemo::StockMarketFeed_var stk_ref;
    try {
        CORBA::Object_var stk_obj = ins->resolve_str("StockSvc");
        stk_ref = StockDemo::StockMarketFeed::_narrow(stk_obj);
    }
    catch (CORBA::Exception &ex) {
        cerr << "Could not resolve/narrow the stock_svc IOR from "
             << "the Naming Service." << endl;
        cerr << "Exception info: " << ex << endl;
        return 1;
    }

    double curr_price;

    try {
        curr_price = stk_ref->read_stock(argv[1]);
    }
    catch (StockDemo::StockSymbolNotFound &ex) {
        cerr << "Stock symbol not found: " << argv[1] << endl;
        cerr << "Try another stock symbol." << endl;
        return 1;
    }
    catch (CORBA::Exception &ex) {
        cerr << "Exception received: " << ex << endl;
        return 1;
    }

    cout << argv[1] << " stock price is " << curr_price << endl;
    return 0;
}
```

19 Persistent State Service

The persistent state service (PSS) is a CORBA service for building CORBA servers that access persistent data.

PSS is tightly integrated with the IDL type system and the object transaction service (OTS). Orbix PSS implements the standard `CosPersistentState` module, and adds proprietary extensions in the `IT_PSS` module. PSS's close integration with OTS facilitates the development of portable applications that offer transactional access to persistent data such as a database system.

Writing a CORBA application that uses PSS is a three-step process:

- Define the data in PSDL (persistent state data language), which is an extension of IDL, then run the IDL compiler on the PSDL files to generate C++ code.
- Write a server program that uses PSS to access and manipulate persistent data.
- Set PSS plug-in variables in the application's configuration as required.

Defining Persistent Data

When you develop an application with PSS, you describe datastore components in the persistent state definition language—PSDL—and save these in a file with a `.psdl` extension.

PSDL is a superset of IDL. Like IDL, PSDL is a declarative language, and not a programming language. It adds new keywords but otherwise conforms to IDL syntax conventions. A PSDL file can contain any IDL construct; and any local IDL operation can accept parameters of PSDL types.

Reserved Keywords

The file `CosPersistentState.psdl` contains all PSDL type definitions, and is implicitly included in any PSDL specification. The following identifiers are reserved for use as PSDL keywords (asterisks indicate keywords reserved for use in future PSS implementations). Avoid using any of the following keywords as user-defined identifiers:

```
as*
catalog*
factory
implements
key
of
primary
provides*
ref
scope*
storagehome
storagetype
stores*
strong*
```

Datastore Model

PSDL contains several constructs that you use to describe datastore components. These include:

- `storagetype` describes how data is organized in storage objects of that type.
- `storagehome` describes a container for storage objects. Each storage home is defined by a storage type and can only contain storage objects of that type. Storage homes are themselves contained by a datastore, which manages the data—for example a database, a set of files, or a schema in a relational database. A datastore can contain only one storage home of a given storage type.

Within a datastore, a storage home manages its own storage objects and the storage objects of all derived storage homes.

For example, the following two PSDL files describe a simple datastore with a single `Account` storage type and its `Bank` storage home:

```
// in bank_demo_store_base.psdl
#include<BankDemo.idl>

module BankDemoStoreBase {
    abstract storagetype AccountBase {
        state BankDemo::AccountId account_id;
        state BankDemo::CashAmount balance;
    };

    abstract storagehome BankBase of AccountBase {
        key account_id;
        factory create(account_id, balance);
    };
};


// in bank_demo_store.psdl

#include <BankDemo.idl>
#include <BankDemoStoreBase.psdl>

module BankDemoStore {
    storagetype Account implements BankDemoStoreBase::AccountBase
    {
        ref(account_id);
    };

    storagehome Bank of Account
        implements BankDemoStoreBase::BankBase
    {};
};
```

Abstract Types and Implementations

In the PSDL definitions shown previously, abstract types and their implementations are defined separately in two files:

- `BankDemoStoreBase.psdl` file defines the abstract storage type `AccountBase` and abstract storage home `BankBase`. Abstract storage types and abstract storage homes are abstract specifications, like IDL interfaces.

- `BankDemoStore.psdl` defines the storage type and storage home implementations for `AccountBase` and `BankBase` in `Account` storage type and `Bank` storage home, respectively.

A storage type implements one or more abstract storage types. Similarly, a storage home can implement any number of abstract storage homes. By differentiating abstract types and their implementations, it is possible to generate application code that is independent of any PSS implementation. Thus, it is possible to switch from one implementation to another one without recompiling and relinking.

Given the separation between abstract types and their implementations, the IDL compiler provides two switches for processing abstract and concrete definitions:

- The `-psdl` switch compiles abstract definitions. For example:

  ```
  idl -psdl bank_demo_store_base.psdl
  ```

  The IDL compiler generates a C++ abstract base class for each `abstract storagetype` and `abstract storagehome` that is defined in this file.

- The `-pss_r` switch generates C++ code that maps concrete PSDL constructs to relational and relational-like database back-end drivers. For example:

  ```
  idl -pss_r bank_demo_store.psdl
  ```

  The IDL compiler generates C++ classes for each `storagetype` and `storagehome` that is defined in this file.

Note: If you maintain all PSDL code in a single file, you should compile it only with the `-pss_r` switch.

Defining Storage Objects

A storage object can have both state and behavior. A storage object's abstract storage type defines both with state members and operations, respectively.

Syntax

The syntax for an abstract storage type definition is similar to the syntax for an IDL interface; unlike an interface, however, an abstract storage type definition cannot contain constants or type definitions.

You define an abstract storage type with this syntax:

```
abstract storagetype abstract-storagetype-name
    [: base-abstract-storage-type[,...]
{
    [ operation-spec; ]...
    [ state-member-spec; ]...
};
```

For example:

```
abstract storagetype AccountBase {
    state BankDemo::AccountId account_id;
    state BankDemo::CashAmount balance;
};
```

The following sections discuss syntax components in greater detail.

Inheritance

As with interfaces, abstract storage types support multiple inheritance from base abstract storage types, including diamond-shape inheritance. It is illegal to inherit two members (state or operation) with the same name.

State Members

A storage object's state members describe the object's data; you can qualify a state member with the `readonly` keyword. You define a state member with the following syntax:

```
[readonly] state type-spec member-name;
```

For each state member, the C++ mapping provides accessor functions that get and set the state member's value (see page 459).

A state member's type can be any IDL type, or an abstract storage type reference.

Operations

Operations in an abstract storage type are defined in the same way as in IDL interfaces. Parameters can be any valid IDL parameter type or abstract storage type reference.

Inherited Operations

All abstract storagetypes implicitly inherit from `CosPersistentState::StorageObject`

```
module CosPersistentState {

    // ...
    native StorageObjectBase;

    abstract storagetype StorageObject {
        void destroy_object();
        boolean object_exists();
        Pid get_pid();
        ShortPid get_short_pid();
        StorageHomeBase get_storage_home();
    };
};
```

You can invoke StorageObject operations on any incarnation of a storage object; they are applied to the storage object itself:

destroy_object() destroys the storage object.

object_exists() returns true if the incarnation represents an actual storage object.

get_pid() and get_short_pid() return the storage object's `pid` and `short-pid`, respectively.

get_storage_home() returns the storage home instance that manages the target storage object instance.

Forward Declarations

As with IDL interface definitions, PSDL can contain forward declarations of abstract storage types. The actual definition must follow later in the PSDL specification.

Defining Storage Homes

You define an abstract storage home with an `abstract storagehome` definition

```
abstract storagehome storagehome-name of abstract-storage-type
{
    [ key-specification ]
    [ factory operation-name( state-member[, ...] ); ]
};
```

For example, the following PSDL defines abstract storage home `BankBase` of storage type `AccountBase`:

```
abstract storagehome BankBase of AccountBase
{
    key account_id;
    factory create(account_id, balance);
};
```

A storage home lacks state but it can have behavior, which is described by operations that are defined in its abstract storage homes. For example, you locate and create a storage object by calling operations on the storage home where this object is stored.

All storage home instances implicitly derive from local interface `CosPersistentState::StorageHomeBase`:

```
module CosPersistentState {
    exception NotFound {};
    native StorageObjectBase;

    // ...
    local interface StorageHomeBase {

        StorageObjectBase
        find_by_short_pid(
            in ShortPid short_pid
```

```
        ) raises (NotFound);
    };
};
```

`find_by_short_pid()` looks for a storage object with the given short pid in the target storage home. If the search fails, the operation raises exception `CosPersistentState::NotFound`.

Keys

An abstract storage home can define one key. A key is composed from one or more state members that belong to the storage home's abstract storage type, either directly or through inheritance. This key gives the storage home a unique identifier for the storage objects that it manages.

Two IDL types are not valid for use as key members: `valuetype` and `struct`.

A key declaration implicitly declares a pair of finder operations; for more information, see page 427.

Simple Keys

A simple key is composed of a single state member. You declare a simple key as follows:

key *key-name* (*state-member*);

For example, the PSDL shown earlier defines abstract storage home `BankBase` for storage objects of abstract type `AccountBase`. This definition can use state member `account_id` to define a simple key as follows:

```
key accno(account-id);
```

If the key's name is the same as its state member, you can declare it in this abbreviated form:

```
key account-id;
```

Composite Keys

A composite key is composed of multiple state members. You declare a composite key as follows:

key *key-name* (*state-member, state-member[,...)*

A composite key declaration must specify a key name. The types of all state members must be comparable. The following types are comparable:

- integral types: `octet`, `short`, `unsigned short`, `long`, `unsigned long`, `long long`, `unsigned long long`
- fixed types
- `char`, `wchar`, `string`, `wstring`
- `sequence<octet>`
- `struct` with only comparable members

Finder Operations

A key declaration is equivalent to the declaration of two PSDL finder operations that use a given key to search for a storage object among the storage objects that are managed directly or indirectly by the target storage home:

find_by_*key-name*() returns an incarnation of the abstract storage home's abstract storage type:

```
abstract-storagetype find_by_key-name(parameter-list)
    raises (CosPersistentState::NotFound);
```

find_ref_by_*key_name*() • returns a reference to this storage object:

```
ref<abstract-storage-type> find_ref_by_key_name(parameter-list);
```

Both operations supply a *`parameter-list`* that is composed of `in` parameters that correspond to each state member in the key declaration, listed in the same order. If a storage object with the given key is not found, `find_by_key_name()` raises the `CosPersistentState::NotFound` exception, and `find_ref_by_key_name()` returns a `NULL` reference.

For example, given the following abstract storage type and storage home definitions:

```
abstract storagetype AccountBase {
    state BankDemo::AccountId account_id;
    state BankDemo::CashAmount balance;
};

abstract storagehome Bank of AccountBase {
    key accno(account_id);
    // ...
};
```

The accno key declaration implicitly yields these two PSDL operations:

```
Account find_by_accno(in BankDemo::AccountId)
    raises (CosPersistentState::NotFound);

ref<Account> find_ref_by_accno(in BankDemo::AccountId);
```

Finder operations are polymorphic. For example, the `find_by_accno` operation can return a `CheckingAccount` that is derived from `Account`.

Operations

Each parameter of a local operation can be of a valid IDL parameter type, or of an abstract PSDL type.

Factory Operations

In the PSDL shown earlier, abstract storage home `BankBase` is defined with the `factory create` operation. This operation provides a way to create `Account` objects in a bank, given values for `account_id` and `balance`.

```
abstract storagehome Bank of AccountBase {
    key accno(account_id);
    factory create(account_id, balance);
};
```

Each parameter that you supply to a factory create operation must be the name of a state member of the abstract storage home's abstract storage type, including inherited state members.

The definition of a factory operation is equivalent to the definition of the following operation:

`abstract-storage-type factory-op-name``(`*`parameter-list`*`);`

where *`parameter-list`* is composed of `in` parameters that correspond to each state member in the factory operation declaration, listed in the same order.

For example, given this factory declaration:

```
abstract storagetype AccountBase {
    state BankDemo::AccountId account_id;
    state BankDemo::CashAmount balance;
};

abstract storagehome Bank of AccountBase {
```

```
    // ...
    factory create(account_id, balance);
};
```

The `create` factory declaration implicitly yields this operation, which uses conventional IDL-to-C++ mapping rules:

```
Account create(
    in BankDemo::AccountId account_id,
    in BankDemo::CashAmount balance
);
```

Inheritance

An abstract storage home can inherit from one or more abstract storage homes, and support diamond-shape inheritance. The following constraints apply to a base abstract storage home:

- The base abstract storage homes must already be defined.
- The base abstract storage homes must use the same abstract storage type or base abstract storage type as the derived abstract storage home.
- An abstract storage home cannot inherit two operations with the same name.

Forward Declarations

As with IDL interface definitions, PSDL can contain forward declarations of abstract storage homes.

Implementing Storage Objects

A storage type implements one or more abstract storage types, and can inherit from one other storage type. Storage type implementations are defined as follows:

```
storagetype storagetype-name [: storagetype-name ]
    implements abstract-storagetype[, abstract-storagetype]...
{
    [ state-member-spec; ]...
    [ ref(state-member[, state-member]...) ]

};
```

The implemented abstract storage type *abstract_storagetype* must specify a previously defined abstract storage type.

State Members

A storage type can define state members; these state members supplement any state members in the abstract storage types that it implements, or that it inherits from other implementations. You define a state member with the following syntax:

```
[readonly] state type-spec member-name;
```

Reference Representation

A storage type can define a reference representation that serves as a unique identifier for storage objects in a storage home of that storage type. A storage type without any base storage type can define a single reference representation, which is composed of one or more state members. For example:

```
storagetype Account implements BankDemoStoreBase::AccountBase
{
    ref(account_id);
};
```

The state members that compose a reference representation must be defined either in:

- One of the abstract storagetypes that this storage type directly implements
- The current storage type

Implementing Storage Homes

A storage home implements one or more previously defined abstract storage homes with this syntax:

```
storage-home storagehome-name[ : storagehome-name]
    of storagetype [ implements abstract-storagehome[,...] ]
{
    [primary-key-spec];
};
```

A storage home specification must include these elements:

- A storage type that derives from the base storage home's storage type. The storage home's storage type must implement the abstract storage type of each of the implemented abstract storage homes.
- Either inherits an existing storage home, or implements one or more existing abstract storage home.

Inheritance

A storage home can inherit form a previously defined storage home. The following constraints apply:

- The storage type of the base storage home must be a base of the storage home's own storage type.
- Two storage homes in a storage home inheritance tree cannot have the same storage type.

For example, the following specification is not legal:

```
storagetype A {/* ... */};
storagetype B : A {/* ... */};
storagehome H of A {};
storagehome H2 of B : H {};
storagehome H3 of B : H {}; // error -- B is already storagetype
                            // of another sub-storage-home of H
```

Primary Key Declaration

A primary key declaration specifies a distinguished key, as implemented in relational systems. You can define a primary key in any storage home without a base storage home.

You can define a primary key in two ways:

- `primary key` *`key-spec`*

  `key-spec` denotes a key that is declared in one of the implemented abstract storagehomes.

- `primary key ref`

  This statement tells the PSS implementation to use the state members of the reference representation as the primary key.

Accessing Storage Objects

You access a storage object through its language-specific implementation, or storage object *incarnation*. A storage object incarnation is bound to a storage object in the datastore and provides direct access to the storage object's state. Thus, updating a storage object incarnation also updates the corresponding storage object in the datastore.

Likewise, to use a storage home, you need a programming language object, or storage home *instance*.

To access a storage object, a server first obtains a logical connection between itself and the datastore that contains this storage home's storage object. This logical connection, or *session*, can offer access to one or more datastores.

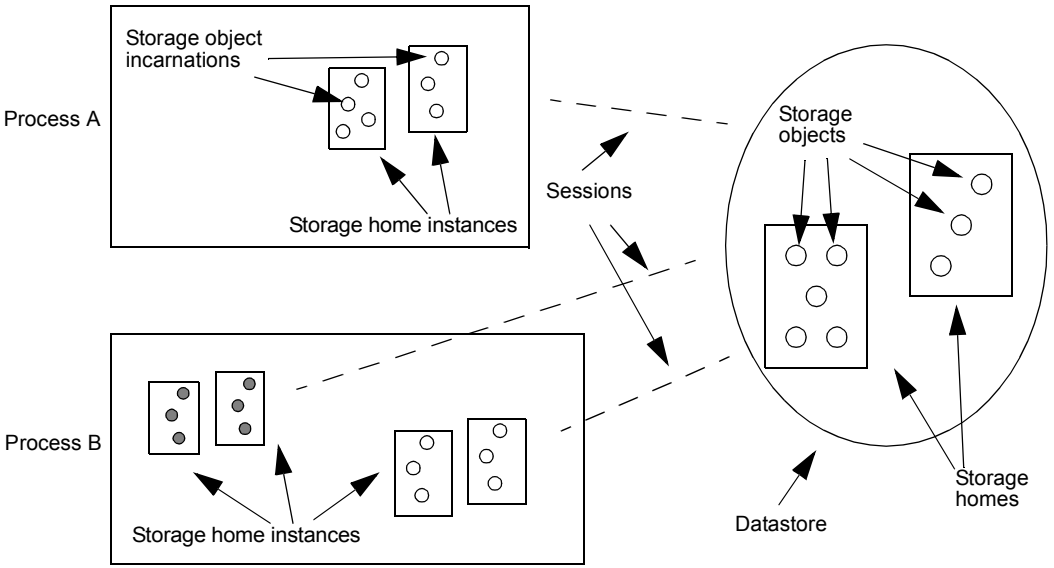

Figure 40: *A server process uses sessions to establish a logical connection with a datastore and its contents*

Creating Transactional Sessions

PSS provides a local connector object that you use to create sessions. Because PSS is designed for use in transactional servers, Orbix provides its own session manager, which automatically creates transactional sessions that can be associated with transactions. You can also manage transactional sessions explicitly.

In either case, you create transactional sessions in these steps:

1. Get a reference to the transaction service's current object by calling `resolve_initial_references("TransactionCurrent")` on the ORB, then narrow the returned reference to a `CosTransactions::Current` object.
2. Get a reference to a connector object by calling `resolve_initial_references("PSS")` on the ORB, then narrow the returned reference to a connector object:
   - `IT_PSS::Connector` object to use an Orbix SessionManager.
   - `CosPersistentState::Connector` to use standard PSS transactional sessions.
3. Create storage object and storage home factories and register them with a Connector object. This allows PSS to create storage object incarnations and storage home instances in the server process, and thereby enable access to the corresponding datastore objects.

   For each PSDL storage home and storage object implementation, the IDL compiler, using the `-pss_r` switch, generates a factory creation and registration operation.For example, given a PSDL storage home definition of `BankDemoStore::Bank`, you can instantiate its storage home factory as follows:

   ```
   CosPersistentState::StorageHomeFactory* bank_factory =
       new IT_PSS_StorageHomeFactory<BankDemoStore::Bank>;
   ```
4. After registering factories with the connector, the connector assumes ownership of the factories, so the server code should call `_remove_ref()` on each factory object reference to avoid memory leaks.
5. Create transactional sessions. You can do this in two ways:

- Create an Orbix SessionManager, which creates and manages the desired number of sessions.
- Create standard PSS TransactionalSession objects.

6. Associate sessions with transactions. How you do so depends on whether you create sessions with a SessionManager or with standard PSS operations:
   - You associate a SessionManager's sessions with transactions through `IT_PSS::TxSessionAssociation` objects.
   - You associate standard transactional sessions with transactions through the TransactionalSession object's `start()` operation.

The following example shows how a server can implement steps 1-4. This code is valid whether you use an Orbix SessionManager or standard PSS transactional sessions.

```
int
main(int argc, char** argv)
{
    // ...
    try
    {
        // Initialise the ORB as configured in the IMR

    cout << "Initializing the ORB" << endl;
    global_orb = CORBA::ORB_init( argc, argv, "demos.pss.bank" );

1   CORBA::Object_var obj =
    global_orb->resolve_initial_references("TransactionCurrent");

    CosTransactions::Current_var tx_current =
        IT_PSS::Connector::_narrow( obj );
    assert(!CORBA::is_nil(tx_current));
2   CORBA::Object_var obj =
        global_orb->resolve_initial_references("PSS");

    IT_PSS::connector_var connector =
        IT_PSS::Connector::_narrow( obj );
    assert(!CORBA::is_nil(connector));

3   // Create and register storage object and
    // storage home factories
```

```
CosPersistentState::StorageObjectFactory *acct_factory =
    new IT_PSS::StorageObjectFactory<BankDemoStore::Account>;

CosPersistentState::StorageHomeFactory *bank_factory =
    new IT_PSS::StorageHomeFactory<BankDemoStore::Bank>;

connector->register_storage_object_factory(
    BankDemoStore::_tc_Account->id(),
    acct_factory
);

connector->register_storage_home_factory(
    BankDemoStore::_tc_Bank->id(),
    bank_factory
);

4   // after registration, connector owns factory objects,
    // so remove factory references from memory

    acct_factory->_remove_ref();
    bank_factory->_remove_ref();

    // ...
    // continuation depends on whether you use Orbix SessionManager
    // or PSS TransactionalSessions
    //...
```

The sections that follow describe the different ways to continue this code, depending on whether you use a SessionManager or standard PSS transactional sessions.

Using the SessionManager

After you create and register storage object and storage home factories, you create a SessionManager and associate transactions with its sessions as follows:

1. Set a list of parameters for the SessionManager to be created, in a `CosPersistentState::ParameterList`. At a minimum, the parameter list specifies the `Resource` that sessions connect to—for example, a Berkeley DB environment name. It can also specify the number of

sessions that are initially created for the SessionManager, and whether to add sessions when all sessions are busy with requests.Table 24 describes all parameter settings.

2. Create a SessionManager by calling `it_create_session_manager()` on the Orbix connector. The SessionManager always creates at least two transactional sessions:
   - A shared read-only session for read-only non-transactional requests.
   - A pool of read-write serializable transactional sessions for write requests, and for any request that is executed in the context of a distributed transaction.
3. Create an association object `IT_PSS::TxSessionAssociation` to associate the SessionManager and the transaction.
4. Use the association object to perform transactional operations on the datastore's storage objects.

The following code implements these steps:

```
// Create SessionManager with one read-only read-committed
// multi-threaded transactional session and one read-write
// serializable single-threaded transactional session

1   CosPersistentState::ParameterList parameters(2);
    parameters.length(2);
    parameters[0].name = CORBA::string_dup("to");
    parameters[0].val <<= CORBA::Any::from_string
        ("bank", true);
    parameters[1].name = CORBA::string_dup("single writer");
    parameters[1].val <<= CORBA::Any::from_boolean(true);

2   IT_PSS::SessionManager_var session_mgr =
        connector->it_create_session_manager(parameters);

3   IT_PSS::TxSessionAssociation association(
        session_mgr.in(),
        CosPersistentState::READ_ONLY,
        CosTransactions::Coordinator::_nil() // use the shared
                                             // read-only session
    );

4   // show balances in all accounts
    // The query API is proprietary; it is similar to JDBC
```

```
    IT_PSS::Statement_var statement =
        association.get_session_nc()->it_create_statement();

    IT_PSS::ResultSet_var result_set = statement->execute_query(
        "select ref(h) from PSDL:BankDemoStore/Bank:1.0 h"
    );

    cout << "Listing database: account id, balance" << endl;
    BankDemoStore::AccountBaseRef     account_ref;
    CORBA::Any_var                    ref_as_any;
    while (result_set->next())
    {
        ref_as_any = result_set->get(1);
        CORBA::Boolean ok = (ref_as_any >>= account_ref);
        assert(ok);
        cout << "               "
             << account_ref->account_id()
             << ", $" << account_ref->balance()
             << endl;
    }
    result_set->close();

    association.suspend();
    // ...
    return 0;
}
```

Setting SessionManager Parameters

You supply parameters to `it_create_session_manager()` through a `CosPersistentState::ParameterList`, which is defined as a sequence of `Parameter` types. Each `Parameter` is a `struct` with `name` and `val` members:

- `name` is a string that denotes the parameter type.
- `val` is an `any` that sets the value of `name`.

The parameter list must specify the Resource that sessions connect to—for example, an ODBC datasource name or Oracle database name. Table 24 describes all parameter settings

Table 24: *SessionManager parameters*

Parameter name	Type	Description
to	string	Identifies the datastore to connect to. For example with PSS/DB, it will be an environment name. You must set this parameter.
rw pool size	long	Initial size of the pool of read-write transactional sessions managed by the session manager. The value must be between 1 and 1000, inclusive. The default value is 1.
grow pool	boolean	If set to TRUE, specifies to create a new session to process a new request when all read-write transactional sessions are busy. A value of FALSE, specifies to wait until a read-write transactional session becomes available. The default value is FALSE.
single writer	boolean	Can be set to TRUE only if rw pool size is 1. In this case, specifies to create a single read-write transactional session that allows only one writer at a time. The default value is FALSE.

Creating a SessionManager

You create a SessionManager by calling `it_create_session_manager()` on the Orbix connector. `it_create_session_manager()` takes a single `ParameterList` argument (see page 437), and is defined in the `IT_PSS::Connector` interface as follows:

```
module IT_PSS {
    // ...
    local interface Connector : CosPersistentState::Connector
    {
        SessionManager
        it_create_session_manager(
            in CosPersistentState::ParameterList parameters
        );
    };
}
```

Associating a Transaction with a Session

The association object `IT_PSS::TxSessionAssociation` associates a transaction with a session that is managed by the SessionManager. You create an association object by supplying it with a SessionManager and the access mode. The `CosPersistentState` module defines two `AccessMode` constants: `READ_ONLY` and `READ_WRITE`

The `IT_PSS::TXSessionAssociation` interface defines two constructors for a `TxSessionAssociation` object:

```
namespace IT_PSS {
    //...
    class TxSessionAssociation {
    public:

        TxSessionAssociation(
            SessionManager_ptr                session_mgr,
            CosPersistentState::AccessMode    access_mode
        ) throw (CORBA::SystemException);

        TxSessionAssociation(
            SessionManager_ptr                session_mgr,
            CosPersistentState::AccessMode    access_mode,
            CosTransactions::Coordinator_ptr  tx_coordinator
```

```
        ) throw (CORBA::SystemException);

        ~TxSessionAssociation()
        throw(CORBA::SystemException);
        // ...
};
```

The first constructor supplies only the session manager and access mode. This constructor uses the default coordinator object that is associated with the current transaction (`CosTransactions::Current`). The second constructor lets you explicitly specify a coordinator; or to specify no coordinator by supplying `_nil()`. If you specify `_nil()`, the association uses the shared transaction that is associated with the shared read-only session; therefore, the access mode must be `READ_ONLY`.

A new association is initially in an active state—that is, it allows transactions to use the session to access storage objects. You can change the association's state by calling `suspend()` or `end()` operations on it (see page 441).

Association Object Operations

An association object has several operations that are defined as follows:

```
namespace IT_PSS {
    // ...
    class TxSessionAssociation{
    public:
        // ...
        TransactionalSession_ptr
        get_session_nc() const throw ();

        CosTransactions::Coordinator_ptr
        get_tx_coordinator_nc() const throw();

        void
        suspend() throw (CORBA::SystemException);

        void
        end(
            CORBA::Boolean success = true
        ) throw (CORBA::SystemException);
    };
};
```

get_session_nc() returns the session for this association object. After you obtain the session, you can access storage objects in the datastore that this session connects to.

get_tx_coordinator_nc() returns the coordinator of this association's transaction.

suspend() suspends a session-Resource association. This operation can raise two exceptions:

- PERSIST_STORE: there is no active association
- INVALID_TRANSACTION: The given transaction does not match the transaction of the Resource actively associated with this session.

end() terminates a session-Resource association. The end operation raises the standard exception PERSIST_STORE if there is no associated Resource, and INVALID_TRANSACTION if the given transaction does not match the transaction of the Resource associated with this session. If the success parameter is FALSE, the Resource is rolled back immediately. Like refresh(), end() invalidates direct references to incarnations' data members.

A Resource can be prepared or committed in one phase only when it is not actively associated with any session. If asked to prepare or commit in one phase when still in use, the Resource rolls back. A Resource (provided by the PSS implementation) ends any session-Resource association in which it is involved when it is prepared, committed in one phase, or rolled back.

Using an Association to Access Storage Objects

You can use an association object to access the data in storage objects. The example shown earlier (see page 435) queries the data in all Account storage objects in the Bank storage home. In order to obtain data from a given storage object, you typically follow this procedure:

Use an association object to get the current session.

Managing Transactional Sessions

The previous section shows how to use the Orbix SessionManager to create and manage transactional sessions. The Orbix SessionManager is built on top of the `CosPersistentState::TransactionalSession` interface. You can use this interface to manage transactional sessions directly.

Note: PSS also provides the `CosPersistentState::Session` interface to manage basic sessions for file-like access. This interface offers only non-transactional functionality whose usefulness is limited to simple applications; therefore, it lies outside the scope of this discussion, except insofar as its methods are inherited by `CosPersistentState::TransactionalSession`.

After you create and register storage object and storage home factories, you create a session and associate transactions with it as follows:

1. Create a TransactionalSession by calling `create_transactional_session()` on a `Connector` object.
2. Activate the transactional session by calling `start()` on it. The transactional session creates a new `CosTransactions::Resource`, and registers it with the transaction.

   For more information about `CosTransactions::Resource` objects, see page 368.
3. Use the session-`Resource` association to perform transactional operations on the datastore's storage objects.

Creating a Transactional Session

Sessions are created through `Connector` objects. A `Connector` is a local object that represents a given PSS implementation.

Each ORB-implementation provides a single instance of the local `Connector` interface, which you obtain through `resolve_initial_references("PSS")` then narrowing the returned reference to a `CosPersistentState::Connector` object. You use the `Connector` object to create a TransactionalSession object by calling `create_transactional_session()` on it:

```
module CosPersistentState {
    // ...

    // forward declarations
    local interface TransactionalSession;
    // ...

    struct Parameter {
        string name;
        any val;
    };

    typedef sequence<Parameter> ParameterList;

    local interface Connector {
        // ...
        TransactionalSession
        create_transactional_session(
            in AccessMode access_mode,
            in IsolationLevel default_isolation_level,
            in EndOfAssociationCallback callback,
            in TypeId catalog_type_name,
            in ParameterList additional_parameters
        );
    };
    // ...
};
```

The parameters that you supply to `create_transactional_session()` define the new session's behavior:

- The access mode for all `Resource` objects to be created by the session. The `CosPersistentState` module defines two `AccessMode` constants:

  `READ_ONLY`
  `READ_WRITE`
- The default isolation level for all `Resource` objects to be created by the session. The `CosPersistentState` module defines four `IsolationLevel` constants:

  `READ_UNCOMMITTED`
  `READ_COMMITTED`
  `REPEATABLE_READ`
  `SERIALIZABLE`

- A callback object to invoke when a session-Resource association ends (see page 444).
- A ParameterList that specifies the datastore to connect to, and optionally other session characteristics (see page 444).

Note: The catalog_type_name parameter is currently not supported. Set it to an empty string.

End-of-Association Callbacks

When a session-Resource association ends, the session might not become available immediately. For example, if the session is implemented with an ODBC or JDBC connection, the PSS implementation needs this connection until the Resource is committed or rolled back.

A session pooling mechanism might want to be notified when PSS releases a session. You can do this by passing a EndOfAssociationCallback local object to the Connector::create_transactional_session operation:

```
module CosPersistentState {
   // ...
   local interface EndOfAssociationCallback {
      void released(in TransactionalSession session);
   };
};
```

ParameterList Settings

You set session parameters in a ParameterList, which is a sequence of Parameter types. Each Parameter is a struct with name and val members:

name is a string that denotes the parameter type.

val is an any that sets the value of name.

The parameter list must specify the Resource that sessions connect to—for example, a Berkeley DB environment name. Table 25 describes all parameter settings

Table 25: *ParameterList settings for a TransactionalSession*

Parameter name	Type	Description
to	string	Identifies the datastore to connect to. For example with PSS/DB, it will be an environment name; with PSS/ODBC a datasource name; with PSS/Oracle, an Oracle database name. You must set this parameter.
concurrent	boolean	If set to TRUE, the session can be used by multiple concurrent threads. The default value is FALSE.
single writer	boolean	Can be set to TRUE only if this session is the only session that writes to this database. A value of TRUE eliminates the risk of deadlock; the cache can remain unchanged after a commit. The default value is FALSE.

Activating a Transactional Session

When you create a transactional session, it is initially in an inactive state—that is, the session is not associated with any Resource. You associate the session with a Resource by calling start() on it, supplying the name of a transaction's coordinator object (see page 447). This function associates the session with a Resource, and registers the Resource with the coordinator's transaction.

A transactional session is associated with one Resource object (a datastore transaction), or with no Resource at all. During its lifetime, a session-Resource association can be in one of three states—active, suspended, or ending—as shown in Figure 41:

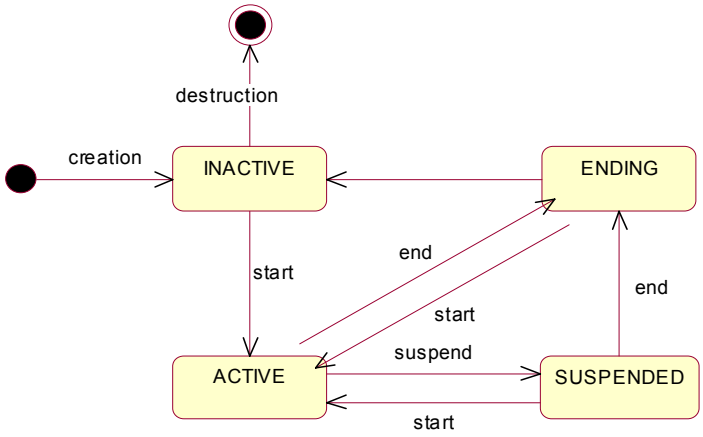

Figure 41: *Transactional session states*

The state members of a storage object's incarnation are accessible only when the transactional session has an active association with a Resource.

Typically, a Resource is associated with a single session for its entire lifetime. However, with some advanced database products, the same Resource can be associated with several sessions, possibly at the same time.

The TransactionalSession interface has this definition:

```
module CosPersistentState {

    // ...
    typedef short IsolationLevel;
    const IsolationLevel READ_UNCOMMITTED  = 0;
    const IsolationLevel READ_COMMITTED    = 1;
    const IsolationLevel REPEATABLE_READ   = 2;
    const IsolationLevel SERIALIZABLE      = 3;
```

```
    interface TransactionalSession : Session {

        readonly attribute IsolationLevel default_isolation_level;

        typedef short AssociationStatus;
        const AssociationStatus NO_ASSOCIATION = 0;
        const AssociationStatus ACTIVE         = 1;
        const AssociationStatus SUSPENDED      = 2;
        const AssociationStatus ENDING         = 3;

        void start(in CosTransactions::Coordinator transaction);
       void suspend(in CosTransactions::Coordinator transaction);
        void end(
               in CosTransactions::Coordinator transaction,
               in boolean success
             );

        AssociationStatus get_association_status();
        CosTransactions::Coordinator get_transaction();
        IsolationLevel
          get_isolation_level_of_associated_resource();
   };
};
```

Managing a Transactional Session

The `TransactionalSession` interface provides a number of functions to manage a transactional session.

start() activates a transactional session. If the session is new, it performs these actions:

- Creates a new `Resource` and registers it with the given transaction.
- Associates itself with this `Resource`.

If the session is already associated with a `Resource` but is in suspended state, `start()` resumes it.

suspend() suspends a session-`Resource` association. This operation can raise two exceptions:

- `PERSIST_STORE`: there is no active association

- INVALID_TRANSACTION: The given transaction does not match the transaction of the Resource actively associated with this session.

end() terminates a session-Resource association. If its success parameter is FALSE, the Resource is rolled back immediately. Like refresh(), end() invalidates direct references to the data members of incarnations.

This operation can raise one of the following exceptions

- PERSIST_STORE: There is no associated Resource
- INVALID_TRANSACTION: The given transaction does not match the transaction of the Resource associated with this session

A Resource can be prepared or committed in one phase only if it is not actively associated with any session. If asked to prepare or commit in one phase when still in use, the Resource rolls back. A Resource ends any session-Resource association in which it is involved when it is prepared, committed in one phase, or rolled back.

Note: In XA terms, start() corresponds to xa_start() with either the TMNOFLAGS, TMJOIN or TMRESUME flag. end corresponds to xa_end() with the TMSUCCESS or the TMFAIL flag. suspend corresponds to xa_end() with the TMSUSPEND or TMSUSPEND | TMMIGRATE flag.

---

get_association_status() returns the status of the association (if any) with this session. The association status can be one of these AssociationStatus constants:

```
NO_ASSOCIATION
ACTIVE
SUSPENDED
ENDING
```

See “Activating a Transactional Session” on page 445 for more information about a transactional session’s different states.

get_transaction() returns the coordinator of the transaction with which the Resource associated with this session is registered. get_transaction returns a nil object reference when the session is not associated with a Resource.

When data is accessed through a transactional session that is actively associated with a Resource, a number of undesirable phenomena can occur:

- Dirty reads: A dirty read occurs when a Resource is used to read the uncommitted state of a storage object. For example, suppose a storage object is updated using Resource 1. The updated storage object's state is read using Resource 2 before Resource 1 is committed. If Resource 1 is rolled back, the data read with Resource 2 is considered never to have existed.
- Nonrepeatable reads: A nonrepeatable read occurs when a Resource is used to read the same data twice but different data is returned by each read. For example, suppose Resource 1 is used to read the state of a storage object. Resource 2 is used to update the state of this storage object and Resource 2 is committed. If Resource 1 is used to reread the storage object's state, different data is returned.

The degree of an application's exposure to these occurrences depends on the isolation level of the Resource. The following isolation levels are defined:

Table 26: *Isolation levels*

Isolation level	Exposure risk
READ_UNCOMMITTED	Dirty reads and the nonrepeatable reads
READ_COMMITTED	Only nonrepeatable reads
SERIALIZABLE	None

Note: Isolation level REPEATABLE_READ is reserved for future use.

get_isolation_level_of_associated_resource() returns the isolation level of the Resource associated with this session. If no Resource is associated with this session, the operation raises the standard exception PERSIST_STORE.

resource_isolation_level (read-only attribute) returns the isolation level of the Resource objects created by this session.

Basic Session Management Operations

The CosPersistentState::TransactionalSession interface inherits a number of operations (via CosPersistentState::Session) from the CosPersistentState::CatalogBase interface. CatalogBase operations provide access to a datastore's storage homes and storage objects; it also provides several memory-management operations:

```
module CosPersistentState {
    interface CatalogBase {
        readonly attribute AccessMode access_mode;

        StorageHomeBase
        find_storage_home(in string storage_home_type_id)
            raises (NotFound);

        StorageObjectBase
        find_by_pid(in Pid the_pid) raises (NotFound);

        void flush();
        void refresh();
        void free_all();
        void close();
    };
    // ...
    local interface Session : CatalogBase {};

    interface TransactionalSession : Session {
    // ...
    };
};
```

find_storage_home() returns a storage home instance that matches the supplied storagehome ID. If the operation cannot find a storage home, it raises a NotFound exception.

find_by_pid() searches for the specified storage object among the storage homes that are provided by the target session. If successful, the operation returns an incarnation of the specified storage object; otherwise, it raises the exception NotFound.

flush() writes to disk any cached modifications of storage object incarnations that are managed by this session. This operation is useful when an application creates a new storage object or updates a storage object, and the modification is not written directly to disk. In this case, you can call `flush()` to rid the cache of "dirty" data.

refresh() refreshes any cached storage object incarnations that are accessed by this session. This operation is liable to invalidate any direct reference to a storage object incarnation's data member.

free_all() sets to 0 the reference count of all PSDL storage objects that have been incarnated for the given session.

PSDL storage object instances are reference-counted by the application. Freeing references can be problematic for storage objects that hold references to other storage objects. For example, if storage object A holds a reference to storage object B, A's incarnation owns a reference count of B's incarnation. When storage objects form a cyclic graph, the corresponding instances own reference count of each other. For example, the following PSDL storage type definition contains a reference to itself:

```
abstract storagetype Person {
    readonly state string full_name;
    state ref<Person> spouse;
};
```

When a couple is formed, each Person incarnation maintains the other Person's incarnation in memory. Therefore, the cyclic graph can never be completely released even if you correctly release all reference counts. In this case, the application must call `free_all()`.

close() terminates the session. When the session is closed, it is also flushed. If the session is associated with one or more transactions (see below) when `close()` is called, these transactions are marked as roll-back only.

Getting a Storage Object Incarnation

After you have an active session, you use this session to get a storage home; you can obtain from this storage home incarnations of its storage objects. You can then use these incarnations to manipulate the actual storage object data.

To get a storage home, call `find_storage_home()` on the session. You narrow the result to the specific storage home type.

Call one of the following operations on the storage home to get the desired storage object incarnation:

- One of the find operations that are generated for key in that storage home. (see page 427).
- `find_by_short_pid()`

Querying Data

Orbix PSS provides simple JDBC-like queries.You use an `IT_PSS::CatalogBase` to create a Statement. For example:

```
IT_PSS::Statement_var stmt
    = catalog->it_create_statement();
```

Then you execute a query that returns a result set:

```
// Gets all accounts
IT_PSS::ResultSet_var result_set
    = stmt->execute_query("select ref(h) from PSDL:Bank:1.0 h");
while (result_set->next())
{
    CORBA::Any_var ref_as_any = result_set->get(1);
    BankDemoStore::AccountRef ref;
    ref_as_any >>= ref;
    cout << "account_id: " << ref->account_id()
         << " balance: $" << ref->balance()
         << endl;
}
result_set->close(); // optional in C++
statement->close();  // optional in C++
```

Orbix PSS supports the following form of query:

```
select ref(h) from home_type_id h
```

The alias must be `h`.

Associating CORBA and Storage Objects

The simplest way to associate a CORBA object with a storage object is to bind the identity of the CORBA object (its `oid`, an octet sequence) with the identity of the storage object.

For example, to make the storage objects stored in storage home `Bank` remotely accessible, you can create for each account a CORBA object whose object ID is the account number (`account_id`).

To make such a common association easier to implement, each storage object provides two external representations of its identity as octet sequences: the `pid` and the `short_pid`:

- `short_pid` is a unique identifier within a storage home and its derived homes.
- `pid` is a unique identifier within the datastore.

Thread Safety

A storage object can be used like a `struct`: it is safe to read concurrently the same storage object incarnation, but concurrent writes or concurrent read/write are unsafe. This behavior assumes that a writer typically uses its own transaction in a single thread; it is rare for an application to make concurrent updates in the same transaction.

Flushing or locking a storage object is like reading this object. Discarding an object is like updating it.

A number of `CosPersistentState::Session` operations are not thread-safe and should not be called concurrently. No thread should use the target session, or any object in the target session such as a storage object incarnation or storage home, when one of these operations is called:

```
Session::free_all()
Session::it_discard_all()
Session::refresh()
Session::close()
TransactionalSession::start()
TransactionalSession::suspend()
TransactionalSession::end()
```

OTS operations are thread-safe. For example one thread can call `tx_current->rollback()` while another thread calls `start()`, `suspend()`, or `end()` on a session involved in this transaction, or while a thread is using storage objects managed by that session.

PSDL Language Mappings

Application code that uses PSS interacts with abstract storage types, abstract storage homes and types defined in the `CosPersistentState` module. This code is completely shielded from PSS-implementation dependencies by the C++ language mapping for abstract storage types, abstract storage homes, and the types defined by the `CosPersistentState` module.

Storage types and storage homes are mapped to concrete programming language constructs with implementation-dependent parts such as C++ members.

The C++ mapping for PSDL and IDL modules is the same. The mapping for abstract storage types and abstract storage homes is similar to the mapping for IDL structs and abstract valuetypes; the mapping for storage types and storage homes is similar to the mapping for IDL structs or valuetypes.

Implementation of operations in abstract storage types and abstract storage homes are typically provided in classes derived from classes generated by the `psdl` backend to the IDL compiler.

The `CosPersistentState` module defines factories to create instances of all user-defined classes, and operations to register them with a given connector:

```
module CosPersistentState {
    native StorageObjectFactory;
    native StorageHomeFactory;
    native SessionFactory;

    interface Connector {

        StorageObjectFactory
        register_storage_object_factory(
           in TypeId storage_type_name,
           in StorageObjectFactory factory
        );
```

```
        StorageHomeFactory
        register_storage_home_factory(
           in TypeId storage_home_type_name,
           in StorageHomeFactory factory
        );

        SessionFactory
        register_session_factory(
           in TypeId catalog_type_name,
           in SessionFactory factory
        );

        // ...
    };
};
```

Each `register_` operation returns the factory previously registered with the given name; it returns `NULL` if there is no previously registered factory.

The `CosPersistentState` module also defines two enumeration types:

```
module CosPersistentState {
   enum YieldRef { YIELD_REF };
   enum ForUpdate { FOR_UPDATE };
};
```

YieldRef defines overloaded functions that return incarnations and references.

ForUpdate defines an overloaded accessor function that updates the state member.

abstract storagehome

The language mappings for abstract storage homes are defined in terms of an equivalent local interface: the mapping of an abstract storage home is the same as the mapping of a local interface of the same name.

Inherited abstract storages homes map to inherited equivalent local interfaces in the equivalent definition.

The equivalent local interface of an abstract storage home that does not inherit from any other abstract storage home inherits from local interface `CosPersistentState::StorageHomeBase`.

abstract storagetype

An abstract storage type definition is mapped to a C++ abstract base class of the same name. The mapped C++ class inherits (with public virtual inheritance) from the mapped classes of all the abstract storage type inherited by this abstract storage type.

For example, given this PSDL abstract storage type definition:

```
abstract storagetype A {}; // implicitly inherits
                           // CosPersistentState::StorageObject
abstract storagetype B : A {};
```

the IDL compiler generates the following C++ class:

```
class  A  :
   public virtual CosPersistentState::StorageObject {};
class ARef { /* ... */};
class B : public virtual A {};
class BRef {/*... */};
```

The forward declaration of an abstract storage type is mapped to the forward declaration of its mapped class and `Ref` class.

Ref Class

For each abstract storage type and concrete storage type definition, the IDL compiler generates the declaration of a concrete C++ class with `Ref` appended to its name.

A `Ref` class behaves like a smart pointer: it provides an `operator->()` that returns the storage object incarnation corresponding to this reference; and conversion operators to convert this reference to the reference of any base type.

Note: `Ref` types manage memory in the same way as `_ptr` reference types. For functionality that is equivalent to a `_var` reference type, the IDL compiler (with the `-psdl` switch) also generates `Ref_var` types (see page 459).

A pointer to a storage object incarnation can be implicitly converted into a reference of the corresponding type, or of any base type. Each reference also has a default constructor that builds a `NULL` reference, and a number of member functions that some implementations might be able to provide without loading the referenced object.

Each `Ref` class has the following public members:

- Default constructor that creates a `NULL` reference.
- Non-explicit constructor takes an incarnation of the target storage type.
- Copy constructor.
- Destructor.
- Assignment operator.
- Assignment operator that takes an incarnation of the target [abstract] storage type.
- `operator->()` that dereferences this reference and returns the target object. The caller is not supposed to release this incarnation.
- `deref()` function that behaves like operator->()
- `release()` function that releases this reference
- `destroy_object()` that destroys the target object
- `get_pid()` function which returns the pid of the target object.
- `get_short_pid()` function which returns the short-pid of the target object.
- `is_null()` function that returns true only if this reference is `NULL`.
- `get_storage_home()` function that returns the storage home of the target object.
- For each direct or indirect base class of the abstract storage type, a conversion operator that converts this object to the corresponding `Ref`.

Each reference class also provides a typedef to its target type, `_target_type`. This is useful for programming with templates.

For example, given this abstract storage type:

```
abstract storagetype A {};
```

the IDL compiler generates the following reference class:

```
class ARef
{
public:
    typedef A _target_type;

    // Constructors
    ARef() throw ();
    ARef( A* target ) throw ();
    ARef( const ARef& ref) throw ();
    // Destructor
    ~ARef() throw ();

    // Assignment operator

    ARef& operator=( const ARef& ref ) throw ();
    ARef& operator=(T* obj) throw ();

    // Conversion operators
    operator CosPersistentState::StorageObjectRef() const throw();

    // Other member functions
    void release() throw ();
    A* operator->() throw (CORBA::SystemException);
    A* deref() throw (CORBA::SystemException);
    void destroy_object() throw (CORBA::SystemException);

    CosPersistentState::Pid*
    get_pid() const throw (CORBA::SystemException);

    CosPersistentState::ShortPid*
    get_short_pid() const throw (CORBA::SystemException);

    CORBA::Boolean is_null() const throw ();

    CosPersistentState::StorageHomeBase_ptr
    get_storage_home() const throw (CORBA::SystemException);

    // additional implementation-specific members
};
```

For operation parameters, Refs are mapped as follows:

PSDL	C++
in ref<S>	SRef
inout ref<S>	SRef&
out ref<S>	SRef_out
(return) ref<S>	SRef

Ref_var Classes

The _var class associated with a _var provides the same member functions as the corresponding `Ref` class, and with the same behavior. It also provides these members:

- The `ref()` function returns a pointer to the managed reference, or 0 if the managed reference is `NULL`.
- Constructors and assignment operators that accept `Ref` pointers.

State Members

Each state member is mapped to a number of overloaded public pure virtual accessor and modifier functions, with the same name as the state member. These functions can raise any CORBA standard exception.

A state member of a basic C++ type is mapped like a value data member. There is no modifier function if the state member is read-only.

For example, the following PSDL definition:

```
// PSDL
abstract storagetype Person {
    state string name;
};
```

is mapped to this C++ class:

```
// C++
class Person : public virtual CosPersistentState::StorageObject {
public:
    virtual const char* name() const = 0;
    virtual void name(const char* s) = 0;  // copies
```

```
    virtual void name(char* s) = 0;        // adopts
    virtual void name(const CORBA::string_var &) = 0;
};
```

A state member whose type is a reference to an abstract storage type is mapped to two accessors and two modifier functions. One of the accessor functions takes no parameter and returns a storage object incarnation, the other takes a `CosPersistentState::YieldRef` parameter and returns a reference. One of the modifier functions takes an incarnation, the other one takes a reference. If the state member is read-only, only the accessor functions are generated.

For example, the following PSDL definition:

```
abstract storagetype Bank;

abstract storagetype Account {
    state long id;
    state ref<Bank> my_bank;
};
```

is mapped to this C++ class:

```
// C++
class Account : public virtual CosPersistentState::StorageObject {
public:
    virtual CORBA::Long id() = 0;
    virtual void id(CORBA::Long l) = 0;
    virtual Bank* my_bank() const= 0;
    virtual BankRef my_bank
        (CosPersistentState::YieldRef yr) const = 0;
    virtual void my_bank(BankRef b) = 0;
};
```

All other state members are mapped to two accessor functions—one read-only, and one read-write—and one modifier function. If the state member is read-only, only the read-only accessor is generated. For example, the following PSDL definition:

```
abstract storagetype Person {
    readonly state string name;
    state CORBA::OctetSeq photo;
};
```

is mapped to this C++ class:

```
// C++
class Person : public virtual CosPersistentState::StorageObject {
public:
    virtual const char* name() = 0;
    virtual const OctetSeq& photo() const = 0;
    virtual OctetSeq& photo(CosPersistentState::ForUpdate fu)
                    = 0;
    virtual void photo(const OctetSeq& new_one) = 0;
};
```

Operation Parameters

Table 27 shows the mapping for parameters of type S and ref<S> (where S is an abstract storage type:.

Table 27: *Mapping for PSDL parameters*

PSDL parameter	C++ parameter
`in S param`	`const S* param`
`inout S param`	`S& param`
`out S param`	`S_out param`
`(return) S`	`(return) S*`

storagetype

A `storagetype` is mapped to a C++ class of the same name. This class inherits from the mapped classes of all the abstract storage types implemented by the storage type, and from the mapped class of its base storage type, if any. This class also provides a public default constructor.

All state members that are implemented directly by the storage type are implemented by the mapped class as public functions.

For example, the following PSDL definition:

```
abstract storagetype Dictionary {
    readonly state string from_language;
    readonly state string to_language;
    void insert(in string word, in string translation);
```

```
    string translate(in string word);
};

// a portable implementation:

struct Entry {
    string from;
    string to;
};
typedef sequence<Entry> EntryList;

storagetype PortableDictionary implements Dictionary {
    state EntryList entries;
};
```

is mapped to this C++ class:

```
// C++
class PortableDictionary : public virtual Dictionary /* ... */ {
public:
    const char* from_language() const;
    const char* to_language() const;
    const EntryList& entries() const;
    EntryList& entries(CosPersistentState::ForUpdate fu);
    void entries(const EntryList&);
    PortableDictionary();
    // ...
};
```

For each storage type, a concrete `Ref` class is also generated. This `Ref` class inherits from the `Ref` classes of all the abstract storage types that the storage type implements, and from the `Ref` class of the base storage type, if any.

The IDL compiler generates `Ref` class declarations for a storage type exactly as it does for an abstract storage type. For more information, see page 456.

storagehome

A `storagehome` is mapped to a C++ class of the same name. This class inherits from the mapped classes of all the abstract storage homes implemented by the storage home, and from the mapped class of its base storage home, if any. This class also provides a public default constructor.

A storage home class implements all finder operations implicitly defined by the abstract storage homes that the storage home directly implements.

The mapped C++ class provides two public non-virtual `_create()` member functions with these signatures:

- A parameter for each storage type state member. This `_create()` function returns an incarnation.
- A parameter for each storage type state member, and a `CosPersistentState::YieldRef` parameter. This `_create()` function returns a reference.

It also provides two public virtual `_create()` member functions with these signatures:

- A parameter for each storage type's reference representation members. This `_create()` function returns an incarnation
- A parameter for each storage type's reference representation members, and a `CosPersistentState::YieldRef` parameter. This `_create()` function returns a reference.

For example, given the following definition of storage home `PortableBookStore`:

```
abstract storagetype Book  {
    readonly state string title;
    state float price;
};
abstract storagehome BookStore of Book {};

storagetype PortableBook implements Book {
    ref(title)
};

storagehome PortableBookStore of PortableBook implements BookStore
   {};
```

the IDL compiler (with the `pss_r` backend) generates the C++ class `PortableBookStore`:

```
// C++
class PortableBookStore : public virtual BookStore /* ... */ {
public:
    virtual PortableBook* _create(const char* title, Float price);
    virtual PortableBook* _create();
```

```
    virtual PortableBookRef _create(
        const char* name,
        Float price,
        CosPersistentState::YieldRef yr
    );
    virtual PortableBookRef _create(
        const char* title,
        CosPersistentState::YieldRef yr
    );
    // ...
};
```

Factory Native Types

Native factory types `StorageObjectFactory`, `StorageHomeFactory`, and `SessionFactory` map to C++ classes of the same names:

```
namespace CosPersistentState {

    template class<T>
    class Factory {
        public:
            virtual T* create()
                        throw (SystemException) = 0;
            virtual void _add_ref() {}
            virtual void _remove_ref() {}
            virtual ~Factory() {}
    };

    typedef Factory<StorageObject> StorageObjectFactory;
    typedef Factory<StorageHomeBase> StorageHomeFactory;
    typedef Factory<Session> SessionFactory;
};
```

20 Event Service

This chapter provides a detailed description of the CORBA event service communications model and describes how Orbix 2000 implements this model.

Orbix 2000 implements the CORBA event service which is defined as part of the CORBAservices specification. This specification defines a model for communications between ORB applications that supplements the direct operation call system that client/server applications normally use.

The CORBAservices specification extends the core CORBA specification with a set of services commonly required in ORB applications. Orbix 2000 supports IIOP for interoperable communications between CORBA implementations. Consequently, any IIOP-compliant ORB can interact with Orbix 2000.

Event Service Basics

The CORBA event service specification defines a model of communication that allows an application to send an event that will be received by any number of objects. The model provides two approaches to initiating event communication. For each of these approaches, event communication can take two forms.

Figure 42 illustrates the standard CORBA model for communication between distributed applications.

In this model, a client application calls an IDL operation on a specified object in a server. The client waits for the call to complete and then receives confirmation of the return status. For any operation call there is a single client and a single server, and each must be available for the call to succeed.

This simple, one-to-one communication model is fundamental to the CORBA architecture. However, some ORB applications need a more complex, indirect communication style. The CORBA event service defines a

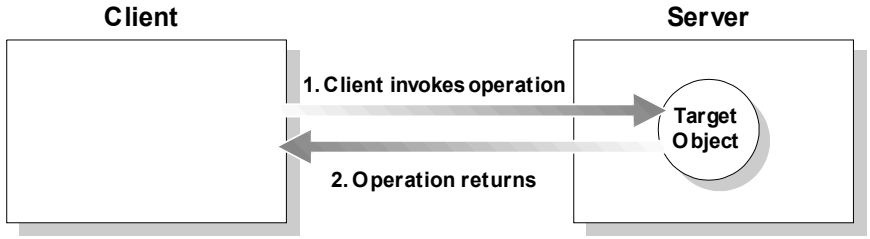

Figure 42: *CORBA model for basic client/server communications*

communication model that allows an application to send a message to objects in other applications without any knowledge about the objects that receive the message.

The CORBA event service introduces the concept of *events* to CORBA communications. An event originates at an event *supplier* and is transferred to any number of event *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 order to support this model, the CORBA event service introduces to CORBA a new architectural element, called an *event channel*. An event channel mediates the transfer of events between the suppliers and consumers as follows:

1. The event channel allows consumers to register interest in events, and stores this registration information.
2. The channel accepts incoming events from suppliers.
3. The channel forwards supplier-generated events to registered consumers.

Suppliers and consumers connect to the event channel and not directly to each other (Figure 43). 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.

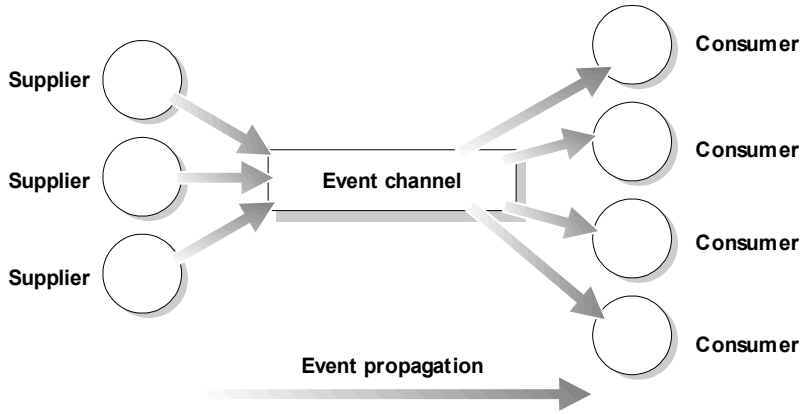

Figure 43: *Suppliers and Consumers Communicating through an Event Channel*

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, and new suppliers and consumers can be easily added to the system. In addition, any supplier or consumer can connect to more than one event channel.

A typical example that uses an event-based communication model is that of a spreadsheet cell. Many documents might be linked to a spreadsheet cell and these documents need to be notified when the cell value changes. However, the spreadsheet software should not need knowledge of each document linked to the cell. When the cell value changes, the spreadsheet software should be able to issue an event which is automatically forwarded to each connected document.

CORBA defines the event service at a level above the ORB architecture. Suppliers, consumers and event channels can be implemented as ORB applications, while events are defined using standard IDL operation calls. Suppliers, consumers and event channels each implement clearly defined IDL interfaces that support the steps required to transfer events in a distributed system.

Figure 44 illustrates an 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

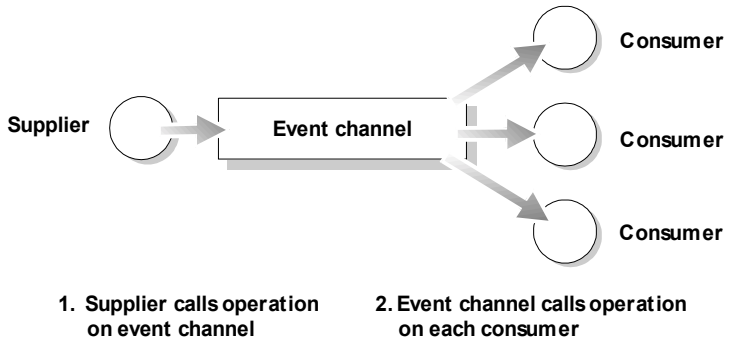

Figure 44: *A sample implementation of event propagation*

occurs when a supplier invokes a clearly defined IDL operation on an object in the event channel application. The event channel propagates the event by invoking a similar operation on objects in each of the consumer servers. To make this possible, the event channel application stores a reference to each of the consumer objects, for example, in an internal list.

This is not the only way in which the concept of events can map to a CORBA system. In particular, the CORBA event service identifies two approaches to initiating the propagation of events, and these affect the implementation architecture. “Initiating Event Communication” on page 468 addresses this topic in detail.

“Types of Event Communication” on page 471 discusses how events can map to IDL operation calls, and describes how you can associate data with an event using IDL operation parameters.

Initiating Event Communication

CORBA specifies two approaches to initiating the transfer of events between suppliers and consumers. These approaches are called the *push model* and the *pull model*. In the push model, suppliers initiate the transfer of events by sending those events to consumers. In the pull model, consumers initiate the transfer of events by requesting those events from suppliers.

This section illustrates each approach in turn, and then describes how these models can be mixed in a single system.

Push Model

In the push model, a supplier generates events and actively passes them to a consumer. In this model, a consumer passively waits for events to arrive. Conceptually, suppliers in the push model correspond to clients in normal CORBA applications, and consumers correspond to servers.

Figure 45 illustrates a push model architecture in which push suppliers communicate with push consumers through an event channel.

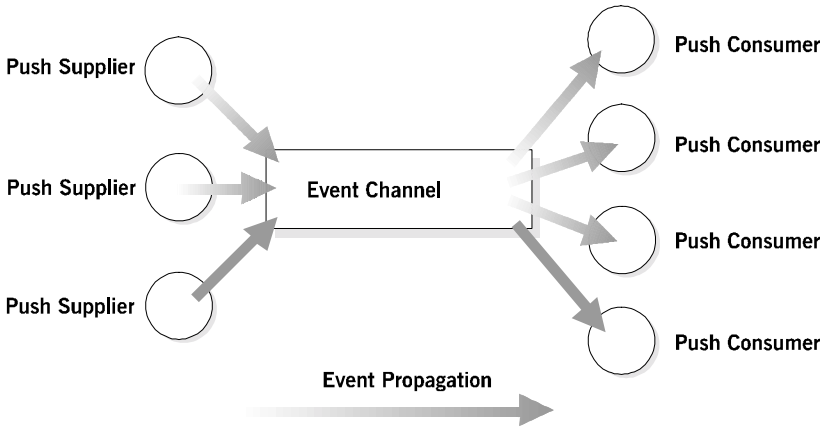

Figure 45: *Push model suppliers and consumers communicating through an event channel*

In this architecture, a supplier initiates the transfer of an event by invoking an IDL operation on an object in the event channel. The event channel invokes a similar operation on an object in each consumer that has registered with the channel.

Pull Model

In the pull model, a consumer actively requests that a supplier generate an event. In this model, the supplier waits for a pull request to arrive. When a pull request arrives, event data is generated by the supplier and returned to the pulling consumer. Conceptually, consumers in the pull model correspond to clients in normal CORBA applications and suppliers correspond to servers.

Figure 46 illustrates a pull model architecture in which pull consumers communicate with pull suppliers through an event channel.

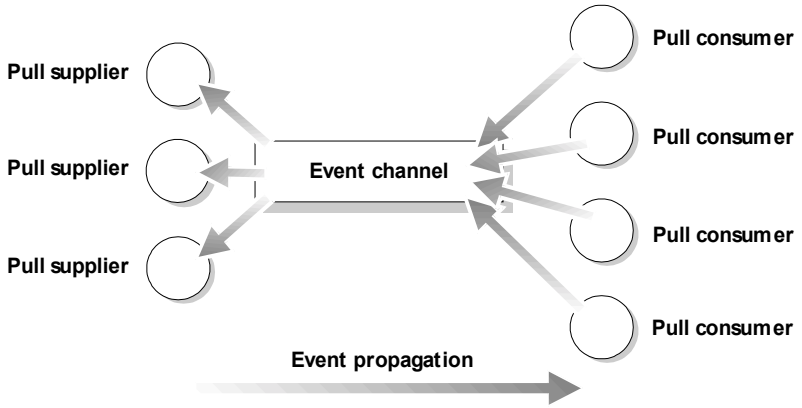

Figure 46: *Pull model suppliers and consumers communicating through an event channel*

In this architecture, a consumer initiates the transfer of an event by invoking an IDL operation on an object in the event channel application. The event channel then invokes a similar operation on an object in each supplier. The event data is returned from the supplier to the event channel and then from the channel to the consumer which initiated the transfer.

Mixing Push and Pull Models in a Single System

Because suppliers and consumers are completely decoupled by an event channel, the push and pull models can be mixed in a single system. For example, suppliers might connect to an event channel using the push model, while consumers connect using the pull model as shown in Figure 47.

In this case, both suppliers and consumers must 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. Unlike the case in which consumers connect using the push model, the event channel takes no initiative in forwarding the event. The event

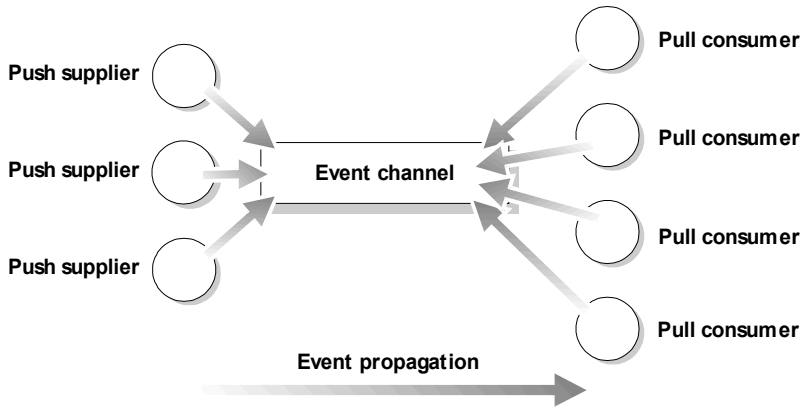

Figure 47: *Push model suppliers and pull model consumers in a single system*

channel stores events supplied by the push suppliers until some pull consumer requests an event, or until a push consumer connects to the event channel.

Types of Event Communication

The CORBA event service maps an event to a successfully completed sequence of operation calls. The operations and the sequence of calls are clearly defined for both push and pull models, and data about an event can be passed as operation parameters or return values. This data is specific to each application and is generally not interpreted by implementations of the CORBA event service, as in Orbix 2000.

The CORBA event service specification defines that event communication can take one of the two forms, *typed* or *untyped*.

Note: The event service implementation in Orbix 2000 supports only untyped communication.

Untyped Event Communication In untyped event communication, an event is propagated by a series of generic `push()` or `pull()` operation calls. The `push()` operation takes a single parameter which stores the event data. The event data parameter is of type any, which allows any IDL defined data type to be passed between suppliers and consumers. The `pull()` operation has no parameters but transmits event data in its return value, which is also of type any. Clearly, in both cases, the supplier and consumer applications must agree about the contents of the any parameter and return value if this data is to be useful.

Typed Event Communication In typed event communication, a programmer defines application-specific IDL interfaces through which events are propagated. Rather than using `push()` and `pull()` operations and transmitting data using an any, a programmer defines an interface that suppliers and consumers use for the purpose of event communication. The operations defined on the interface can contain parameters defined in any suitable IDL data type. In the push model, event communication is initiated simply by invoking operations defined on this interface. The pull model is more complex because event communication is initiated by invoking operations on an interface that is specially constructed from the application-specific interface that the programmer defines.

Programming Interface for Untyped Events

The CORBA event service specification defines a set of interfaces that support the push and pull models of initiating the transfer of events in both typed and untyped format. Orbix 2000 supports only untyped events. This section gives details of the interfaces for these models. The CORBA event service specification defines the roles of consumer, supplier and event channel by describing IDL interfaces that each model must support. The operations on these interfaces allow consumers and suppliers to register with an event channel to enable the propagation of events. The CORBA event service for untyped events also defines a number of administration interfaces that allow suppliers and consumers to register with an event channel to allow the transfer of events between them.

You can find a complete listing of all interfaces relating to the CORBA event service in the *Orbix 2000 Programmer's Reference*.

Registration of Suppliers and Consumers with an Event Channel

A supplier connects to an event channel to indicate that it wishes to transfer events to consumers through that channel. A consumer connects to an event channel to register its interest in any events supplied through that channel. When a supplier or consumer no longer wishes to send or receive events, the application can disconnect itself from the event channel. In some cases, the event channel might need to disconnect a supplier or consumer explicitly.

The CORBA event service defines a set of interfaces that supports untyped event transfer using the push and pull models. These interfaces are described in the remainder of this section.

Push Model for Untyped Events

Four IDL interfaces support connection to and disconnection from event channels using the push model:

`PushSupplier`
`PushConsumer`
`ProxyPushConsumer`
`ProxyPushSupplier`

The interfaces `PushSupplier` and `ProxyPushConsumer` allow suppliers to supply events to an event channel.

The interfaces `PushConsumer` and `ProxyPushSupplier` are specific to consumers, allowing them to receive events from an event channel.

These four interfaces are defined in IDL as follows:

```
module CosEventComm {
    exception Disconnected {
    };

    interface PushConsumer {
        void push (in any data) raises (Disconnected);
        void disconnect_push_consumer ();
    };

    interface PushSupplier {
        void disconnect_push_supplier();
    };
};
```

```
module CosEventChannelAdmin {
    exception AlreadyConnected {
    };

    exception TypeError {
    };

    interface ProxyPushConsumer : CosEventComm::PushConsumer {
        void connect_push_supplier (
            in CosEventComm::PushSupplier push_supplier)
            raises (AlreadyConnected);
    };

    interface ProxyPushSupplier : CosEventComm::PushSupplier {
        void connect_push_consumer (
            in CosEventComm::PushConsumer push_consumer)
            raises (AlreadyConnected, TypeError);
    };

    ...
};
```

Connecting a Supplier A supplier initiates connection to an event channel by obtaining a reference to an object of type `ProxyPushConsumer` in the channel. The supplier application might wish to be notified if the event channel terminates the connection. If so, the supplier then invokes `connect_push_supplier()` on that object, passing a reference to an object of type `PushSupplier` as an operation parameter. If the `ProxyPushConsumer` is already connected to a `PushSupplier`, `connect_push_supplier()` will raise the exception `AlreadyConnected`.

Connecting a Consumer A consumer first obtains a reference to a `ProxyPushSupplier` object implemented in the event channel. In order to register its interest in events from the channel, the consumer then invokes `connect_push_consumer()` on the `ProxyPushSupplier` object. The consumer passes a reference to an object of type `PushConsumer` to the operation call.

If `ProxyPushSupplier` is already connected to a `PushConsumer`, `connect_push_consumer()` will raise the exception `AlreadyConnected`.

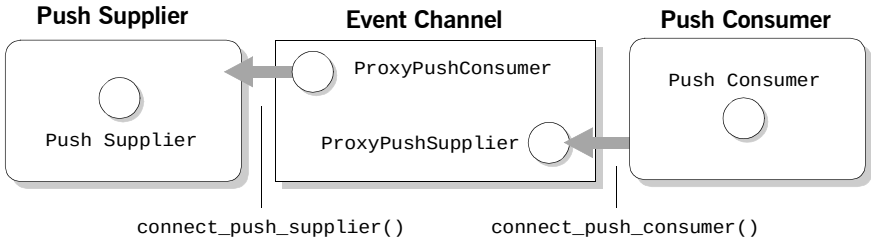

Figure 48: *Push supplier and push consumer connecting to an event channel in the untyped model*

Figure 48 illustrates how a supplier and consumer connect to an event channel. Note that there are no dependencies between the connection of the supplier and the connection of the consumer.

Pull Model for Untyped Events

A similar set of IDL interfaces supports connection to and disconnection from event channels in the pull model. These interfaces are:

```
PullSupplier
PullConsumer
ProxyPullConsumer
ProxyPullSupplier
```

The interfaces `PullConsumer` and `ProxyPullSupplier` allow consumers to request events from an event channel.

The interfaces `PullSupplier` and `ProxyPullConsumer` allow an event channel to request events from suppliers.

The pull model interfaces are defined in IDL as follows:

```
module CosEventComm {
    exception Disconnected {
    };

    interface PullSupplier {
        any pull () raises (Disconnected);
       any try_pull (out boolean has_event) raises (Disconnected);
        void disconnect_pull_supplier();
    };
```

```
    interface PullConsumer {
        void disconnect_pull_consumer ();
    };
};


module CosEventChannelAdmin {
    exception AlreadyConnected {
    };

    exception TypeError {
    };

    interface ProxyPullSupplier : CosEventComm::PullSupplier {
        void connect_pull_consumer (
            in CosEventComm::PullConsumer pull_consumer)
            raises (AlreadyConnected);
    };

    interface ProxyPullConsumer : CosEventComm::PullConsumer {
        void connect_pull_supplier (
            in CosEventComm::PushSupplier pull_supplier)
            raises (AlreadyConnected, TypeError);
    };

    ...
};
```

Connecting a Consumer In the pull model, the transfer of events is initiated by consumers. A consumer initiates connection to an event channel by obtaining a reference to an object of type `ProxyPullSupplier` in the channel. The consumer application might wish to be notified if the event channel terminates the connection. If so, it invokes `connect_pull_consumer()` on the `ProxyPullSupplier` object, passing a reference to an object of type `PullConsumer` as an operation parameter. If the `ProxyPullSupplier` is already connected to a `PullConsumer`, `connect_pull_consumer()` throws exception `AlreadyConnected`.

Connecting a Supplier To connect to an event channel, a pull supplier first obtains a reference to a `ProxyPullConsumer` object implemented in the event channel. The supplier then invokes `connect_pull_supplier()` on the `ProxyPullConsumer` object, passing a reference to an object of type `PullSupplier` as the operation parameter. If the `ProxyPullConsumer` is already connected to a `PullSupplier`, `connect_pull_supplier()` throws exception `AlreadyConnected`.

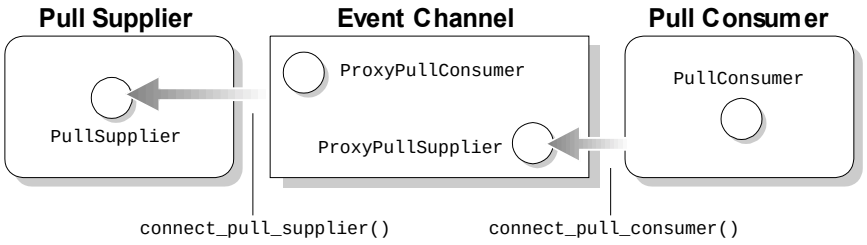

Figure 49: *Pull supplier and pull consumer connecting to an event channel in the untyped model*

Figure 49 illustrates how a pull supplier and pull consumer connect to an event channel. Note that there are no dependencies between the connection of the supplier and the connection of the consumer.

Transfer of Untyped Events Through an Event Channel

The transfer of events from a supplier through an event channel to a consumer follows a simple pattern. Events originate at a supplier. In the push model, a supplier pushes events into the event channel which in turn pushes the events to registered consumers. In the pull model, consumers take the active role by requesting events from the event channel; the event channel, in turn, requests events from registered suppliers. Both methods of transfer are described for *untyped* events in the remainder of this section.

Push Model

The supplier initiates event transfer by invoking `push()` on a `ProxyPushConsumer` object in the event channel, passing the event data as a parameter of type `any`. The event channel then invokes `push()` on the `PushConsumer` object in each registered consumer, again passing the event data as an operation parameter. Conceptually, this transfer is as shown in Figure 50.

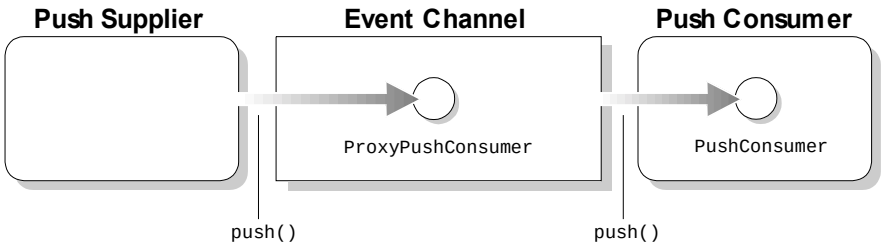

Figure 50: *Transfer of an event through an event channel to a consumer using the untyped push model*

Note that the supplier views the event channel as a single consumer and has no knowledge of the actual consumers. Likewise, the consumer views the event channel as a single supplier. In this way, the channel decouples the supplier and consumer.

Pull Model

The consumer initiates event transfer in the pull model. The consumer initiates event transfer in one of two ways as described below.

pull()

The consumer invokes `pull()` on a `ProxyPullSupplier` object in the event channel. The event channel, if it does not already have an event, invokes `pull()` on the `PullSupplier` object in each registered supplier.

`pull()` blocks until an event is available; the operation then returns the event data in its return value which is of type any. Thus, the consumer application blocks until the event channel can supply an event. The event channel, in turn, blocks until some supplier supplies an event to the channel.

try_pull()

The consumer invokes `try_pull()` on a `ProxyPullSupplier` object in the event channel. The event channel, in turn, invokes `try_pull()` on the `PullSupplier` object in each registered supplier.

If no supplier has an event available, `try_pull()` sets its boolean `has_event` parameter to false and returns immediately. If an event is available from some supplier, `try_pull()` sets the `has_event` parameter to true and returns the event data in its return value which is of type any.

Conceptually, the transfer of an event using the pull model is as shown in Figure 51.

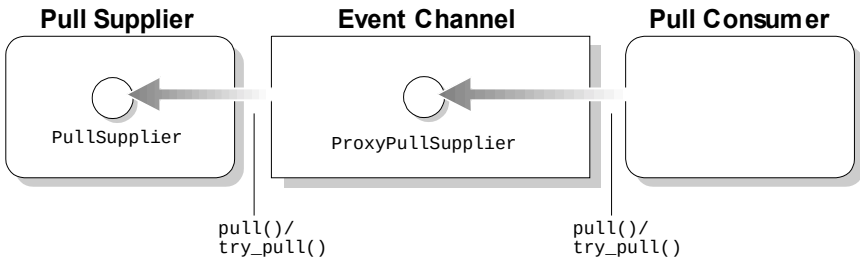

Figure 51: *Transfer of an event through an event channel to a consumer using the untyped pull model*

Note that, as in the push model, the channel decouples suppliers and consumers. The consumer views the event channel as a single supplier and has no knowledge of the actual suppliers. Likewise, the supplier views the event channel as a single consumer.

Event Channel Administration Interfaces

The CORBA event service specification defines a set of interfaces that support event channel administration. These interfaces allow a supplier or consumer to make initial contact with an event channel, and provide a set of standardized operations so that a supplier can obtain a `ProxyPushConsumer` or `ProxyPullConsumer` and a consumer can obtain a `ProxyPushSupplier` or `ProxyPullSupplier` object reference.

Each event channel supports the `EventChannel` interface, which is defined as follows:

```
module CosEventChannelAdmin {
    ...

    interface EventChannel {
        ConsumerAdmin for_consumers ();
        SupplierAdmin for_suppliers ();
        void destroy ();
    };
};
```

If a supplier or consumer wishes to connect to an event channel, it must first obtain a reference to an `EventChannel` object in that channel. It does this by calling `resolve_intital_references()` on "EventService" and narrowing the resulting reference.

A supplier then invokes `for_suppliers()` on the `EventChannel` object. This operation returns a reference to an object of type `SupplierAdmin`, which is defined as follows:

```
module CosEventChannelAdmin {
    interface SupplierAdmin {
        ProxyPushConsumer obtain_push_consumer ();
        ProxyPullConsumer obtain_pull_consumer ();
    };

    ...
};
```

To obtain a reference to a `ProxyPushConsumer` object in the event channel, the supplier invokes `obtain_push_consumer()` on the `SupplierAdmin` object. At this point, the supplier is ready to connect to the channel and begin transferring events using the push model.

The supplier invokes `obtain_pull_consumer()` on the `SupplierAdmin` object if it wishes to obtain a `ProxyPullConsumer`. The supplier is then ready to connect to the channel and to transfer events using the pull model.

Similarly, a consumer invokes `for_consumers()` on an `EventChannel` object in order to obtain a reference to an object of type `ConsumerAdmin`, which is defined as follows:

```
module CosEventChannelAdmin {
    interface ConsumerAdmin {
        ProxyPushSupplier obtain_push_supplier ();
        ProxyPullSupplier obtain_pull_supplier ();
    };

    ...
};
```

If the consumer is using the push model, it then invokes `obtain_push_supplier()` to obtain a reference to a `ProxyPushSupplier`. If the consumer is using the pull model, it invokes `obtain_pull_supplier()` to obtain a reference to a `ProxyPullSupplier` object in the event channel.

The consumer is then free to register its interest in events propagated through the channel.

Overview of the Orbix Event Service

The Orbix event service can implement one or more conceptual event channels. The criteria that determine the number of event channels required by your application architecture are specific to that application. Some applications might transfer each of several event types through a single channel, while others might have multiple channels that act as alternative sources of a single event type.

Figure 52 illustrates a sample architecture where suppliers and consumers communicate through two event channels implemented in a single Orbix server. Note that any given supplier or consumer can connect to multiple event channels simultaneously. In addition, a supplier or consumer can connect to event channels in multiple Orbix servers, if required.

Orbix maintains an `EventChannel` object, a `SupplierAdmin` object and a `ConsumerAdmin` object for each untyped event channel it implements. An ORB application contacts an event channel by obtaining a reference to the

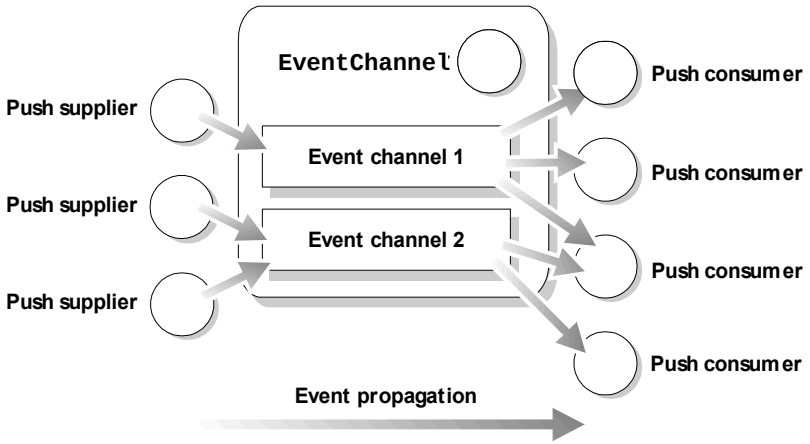

Figure 52: *Sample Orbix event service architecture with two event channels*

corresponding `EventChannel` object. The application then uses this object to retrieve a reference to the `SupplierAdmin` or the `ConsumerAdmin` object, depending on whether the application is a supplier or consumer.

The `SupplierAdmin` object creates and manages `ProxyPushConsumer` objects for a single untyped event channel. For each supplier that connects to the channel, the `SupplierAdmin` creates a `ProxyPushConsumer` object which the supplier can use to generate events. Similarly, the `ConsumerAdmin` object creates and manages a `ProxyPushSupplier` object for each consumer that connects to the event channel.

Components of the Orbix Event Service

An Orbix consumer or supplier is a normal ORB application that communicates with an Orbix server using standard IDL operation calls.

Consequently, the components of your Orbix implementation include the complete IDL definitions for the CORBA event service.

The IDL definitions for the CORBA event service are contained in these files in the `idl` directory:

Table 28: *Orbix Event Service IDL Files*

IDL File	Contents
`CosEventComm.idl`	This file contains the `CosEventComm` module.
`CosEventChannelAdmin.idl`	This file contains the `CosEventChannelAdmin` module.
`event.idl`	This file contains the IT_Event module

Programming with the Untyped Push Model

From a programmer's perspective, the event channel is the key element of a CORBA event service application.

This section describes an ORB application that shows how you can use the Orbix event service to develop push model suppliers and consumers that communicate untyped events through event channels.

Overview of a Sample Application

The example described in this section consists of a push supplier and a push consumer, each of which connects to a single event channel. The supplier repeatedly pushes an event to the event channel and the data associated with each event takes the form of a string. The event channel propagates each event to the consumer, which simply displays the event data. This application is simple, but it illustrates a series of development tasks that apply to all Orbix event service applications.

To develop an Orbix event service application, you must implement the suppliers and consumers as normal ORB applications that communicate with the event channel through IDL interfaces. The Orbix event service fully implements the event channel, which is created in the Orbix event service server application. The IDL definitions for the CORBA event service are supplied with Orbix.

Developing an Untyped Push Supplier

As described in "Transfer of Untyped Events Through an Event Channel" on page 477, a push supplier initiates the transfer of an event by pushing the event into an event channel. The event channel then takes responsibility for forwarding the event to each registered consumer.

This section describes how you can implement a push supplier as an Orbix application that communicates with a single event channel in an Orbix event service server. This application acts as a client to several IDL interfaces implemented in the event channel and acts as a server to the interface `PushSupplier`, which it implements.

There are three main programming steps in developing a push supplier:

1. Obtain a reference for a `ProxyPushConsumer` object from the event channel.

   "Obtaining a ProxyPushConsumer from an Event Channel" on page 484 explains this step in detail.

2. Invoke `connect_push_supplier()` on the `ProxyPushConsumer` object, to connect a `PushSupplier` implementation object to the event channel.

   "Connecting a PushSupplier Object to an Event Channel" on page 485 explains this step.

3. Invoke `push()` on the `ProxyPushConsumer` object to initiate the transfer of each event.

   "Pushing Events to an Event Channel" on page 486 explains this step.

Obtaining a ProxyPushConsumer from an Event Channel

A push supplier needs to obtain a reference for a `ProxyPushConsumer` object in an event channel in order to transfer events to the channel for later distribution to consumers. The supplier transfers events by invoking `push()` on the target `ProxyPushConsumer` object.

In order to obtain a `ProxyPushConsumer` object reference from an event channel, a supplier must implement the following programming steps:

1. Obtain a reference to an `EventChannel`.
2. Invoke `for_suppliers()` on the `EventChannel` object, in order to obtain a `SupplierAdmin` object reference.

3. Invoke obtain_push_consumer() on the SupplierAdmin object. This operation returns a ProxyPushConsumer object reference.

```
// C++
CORBA::Object_var objVar =
   orb->resolve_initial_references("EventService");
IT_Event::EventChannelFactory_var factory =
   IT_Event::EventChannelFactory::_narrow(objVar);
int channel = factory->create_channel("my_channel", id);
CosEventChannelAdmin::SupplierAdmin_var sa =
   channel->for_suppliers();
CosEventChannelAdmin::ProxyPushConsumer_var ppc =
   sa->obtain_push_consumer();
```

Connecting a PushSupplier Object to an Event Channel

When the supplier has retrieved the EventChannel object reference and used this to obtain a ProxyPushConsumer, the supplier needs to connect an implementation of the PushSupplier interface to the event channel. As described in “Registration of Suppliers and Consumers with an Event Channel” on page 473, this interface is defined as follows:

```
module CosEventComm {
    ...

    interface PushSupplier {
        void disconnect_push_supplier ();
    };
};
```

The role of this interface is to allow the event channel to disconnect the supplier by invoking disconnect_push_supplier(). This can happen if the event channel closes down.

```
// C++
// This assumes we have a reference to "RootPOA" and have activated
// this object
CORBA::Object_var obj = poa->servant_to_reference(this);
ref = CosEventComm::PushSupplier::_narrow(obj);
ppc->connect_push_supplier(obj);
```

Here, the supplier connects an object of this type to an event channel by calling connect_push_supplier() on the ProxyPushConsumer object.

Pushing Events to an Event Channel

The following code extract is a simple demonstration of initiating the transfer of events:

```
// C++
while (!pushSupplier.complete())
   {
      if (orb->work_pending())
      {
         orb->perform_work();
      }
    CORBA::Any a;
    a <<= eventDataString;
    ppc->push (a);
}
```

In this example, the supplier repeatedly pushes an event to the event channel by calling `push()` on a `ProxyPushConsumer` object. The supplier represents the event data using a simple string, but this is not necessary in general. `push()` takes a parameter of type any for the event data, so you can represent this data using any IDL type.

Note that the supplier stops sending events only when it receives an incoming `disconnect_push_supplier()` operation call from the event channel. As an alternative, the supplier could explicitly disconnect from the event channel by invoking `disconnect_push_consumer()` on the event channel `ProxyPushConsumer` object.

Push Supplier Application

To see a complete example of how the above steps fit together, take a look at the Event Service demos, in `<orbix_2000_installation_dir>/demos/events/`.

Developing an Untyped Push Consumer

A push consumer receives events from an event channel, with no knowledge of the suppliers from which those events originated. An event channel propagates an event to a push consumer by invoking `push()` on a `PushConsumer` implementation object in the consumer application. As such,

the main functionality of a push consumer is associated with registering a PushConsumer object with an event channel and receiving incoming operation calls on that object.

To develop a push consumer application, you must implement the following steps:

1. Obtain a reference for a ProxyPushSupplier object from the event channel.

   "Obtaining a ProxyPushSupplier from an Event Channel" on page 487 explains this step.

2. Connect a PushConsumer implementation object to the event channel, by invoking connect_push_consumer() on the ProxyPushSupplier object.

   "Connecting a PushConsumer Object to an Event Channel" on page 488 explains this step.

3. Monitor incoming operation calls.

   "Monitoring Incoming Operation Calls" on page 489 explains this step.

Obtaining a ProxyPushSupplier from an Event Channel

Each push consumer connected to an event channel receives every event raised by every supplier connected to the channel. However, consumers have no knowledge of the suppliers. Consumers simply connect to an object in the event channel which acts as a single source of events.

This object is responsible for storing a PushConsumer object reference for each connected consumer and invoking the push() operation on each of these references when a supplier transmits an event. The event channel object which stores consumer references is of type ProxyPushSupplier. The first task in developing a push consumer application is to obtain a reference to this object.

There are three stages in obtaining a ProxyPushSupplier object reference:

1. Obtain a reference to an EventChannel object in the event channel.
2. Invoke for_consumers() on the EventChannel object to obtain a ConsumerAdmin object reference.
3. Invoke obtain_push_supplier() on the ConsumerAdmin object. This operation returns a ProxyPushSupplier object reference.

You can implement the first of these steps in exactly the manner described for push supplier applications in "Obtaining a ProxyPushConsumer from an Event Channel" on page 484. The remaining steps involve normal operation invocations.

Connecting a PushConsumer Object to an Event Channel

When a consumer has obtained a reference to the `ProxyPushSupplier` object in an event channel, the next step is to register a `PushConsumer` implementation object with the `ProxyPushSupplier`. The event channel uses the `PushConsumer` object to propagate events to the consumer.

As described in "Registration of Suppliers and Consumers with an Event Channel" on page 473, the CORBA event service specification defines the interface `PushConsumer` as follows:

```
module CosEventComm {
    exception Disconnected {};

    interface PushConsumer {
        oneway void push (in any data) raises (Disconnected);
        void disconnect_push_consumer ();
    };
    ...
};
```

When an event arrives at an event channel, the channel `ProxyPushSupplier` object invokes `push()` on each connected consumer, passing the event data as an `any` parameter. A consumer can raise a `Disconnected` exception in the implementation of this call to indicate to the channel that consumer is disconnected and the event was propagated erroneously. The `disconnect_push_consumer()` operation allows an event channel to disconnect a consumer, for example if the channel closes down.

```
// C++
PushConsumer_i servant(orb);
CosEventComm::PushConsumer_var consumer = servant.this();
CosEventChannelAdmin::ConsumerAdmin_var ca =
   channel->for_consumers();
CosEventChannelAdmin::ProxyPushSupplier_var pps =
   ca->obtain_push_supplier();
pps->connect_push_consumer(consumer);
}
```

Monitoring Incoming Operation Calls

The main role of the consumer is to receive events from the event channel in the form of IDL operation calls. In the push event model the consumer is in effect a server, and therefore must remain available to the event channel until the event channel explicitly disconnects it. Consequently, the PushConsumer must process events in a separate thread.

```
// C++
PortableServer::POAManager_var poa_manager =
   root_poa->the_POAManager;
poa_manager->activate();
while (!pushConsumer.complete());
{
   if (orb->work_pending()) {
       orb->perform_work();
   }
// Do other things
}
pps->disconnect_push_supplier;
```

If the consumer receives an invocation on `disconnect_push_consumer()`, then the implementation of this operation sets `pcImpl.m_disconnected` to 1 and breaks the consumer's event processing loop. Consequently, the consumer receives all events until the event channel explicitly forces it to disconnect.

As an alternative, the consumer could explicitly disconnect itself from the event channel when it no longer wishes to receive events. The consumer does this by invoking `disconnect_push_supplier()` on the event channel `ProxyPushSupplier` object.

Push Consumer Application

To see a complete example of how the above steps fit together, take a look at the Event Service demos, in `<orbix_2000_installation_dir>/demos/events/`.

Compiling and Running an Event Service Application

You will need to compile the IDL definitions for the event service as well as compile and build your application.

IDL Definitions for the Event Service

The CORBA standard IDL interfaces for CORBA event service suppliers, consumers and event channels are defined in the files `CosEventComm.idl` and `CosEventChannelAdmin.idl in the idl/omg directory and event.idl` in the `idl/orbix` directory of your Orbix installation. These are the contents of the IDL files:

Table 29: *Orbix Event Service IDL Files*

IDL File	Contents
`CosEventComm.idl`	This file contains the `CosEventComm` module.
`CosEventChannelAdmin.idl`	This file contains the `CosEventChannelAdmin` module.
`event.idl`	This contains the IT_Events module

Compiling an Event Service Application

An Orbix event service supplier or consumer application is simply a standard ORB application that communicates with an event channel server through a set of IDL interfaces. In addition, both suppliers and consumers implement IDL interfaces and therefore act as ORB servers.

To compile an Orbix event service application, you should follow the compilation steps described in the *Orbix 2000 Programmer's Guide*. For example, the following steps are required to build an Orbix application that communicates with an event channel:

1. Compile the IDL definitions accessed by your application, including those in the files `cosevents.idl`, `coseventsadmin.idl`, and `event.idl` as described in “IDL Definitions for the Event Service” on page 490.
2. Compile any IDL generated C++ files required by your application.
3. Compile all other C++ source files associated with your application.
4. Link the object files from steps 2 and 3 with the appropriate Orbix libraries.

Running an Orbix Event Service Application

The Orbix event service is installed when Orbix 2000 services are installed. The default service installed uses the ORB name "event".

Before running an Orbix event service application, you must first decide whether you want to use the default event service or create new copies.

If you want to use new copies, you must register them with the Orbix implementation repository. You must name your ORBs hierarchically as children of the default ORB, such as event.event2.

Running your Application

Once you have registered the Orbix event service, you can run your supplier and consumer applications. In the examples in “Programming with the Untyped Push Model” on page 483 the order in which you run the consumer and supplier applications has no effect on the system functionality. You do not need to register the suppliers or the consumers shown here in the implementation repository.

Lifetime of Proxy Objects

The event server creates a new proxy object when requested for one. This object persists until:

1. Disconnect is invoked upon it.
2. The event channel is destroyed.
3. The IIOP connection is closed.

The proxy is destroyed in all these cases. It is not possible perform another invocation on the object after that, including `push()`, `pull()`, `try_pull()`, `connect()`, or `disconnect()`. If an attempt is made to perform an operation on the destroyed proxy, an `INVALID OBJECT REFERENCE` exception is thrown.

If a `PullConsumer` has invoked `pull()` upon a `ProxyPullSupplier`, and meanwhile `disconnect_pull_supplier()` is invoked upon the `ProxyPullSupplier`, the `pull()` throws a `Disconnected` exception some time after (depending on the `pull_prod_interval` configuration value).

If you attempt to connect an invalid object to a proxy object (where an exception other than `INVALID OBJECT REFERENCE` is thrown), the proxy is not destroyed.

21 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.

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.

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 53 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.

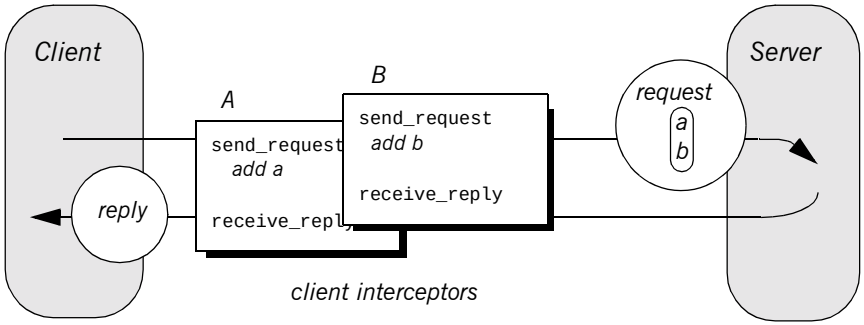

Figure 53: *Client interceptors allow services to access outgoing requests and incoming replies.*

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 502).
- A `PortableInterceptor::ClientRequestInfo` object is supplied to all `ClientRequestInterceptor` interception points (see page 512).
- A `PortableInterceptor::ServerRequestInfo` object is supplied to all `ServerRequestInterceptor` interception points (see page 520).

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 504.

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:

```
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:
  1. Obtain a unique VSCID from the OMG
  2. 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 499).

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.

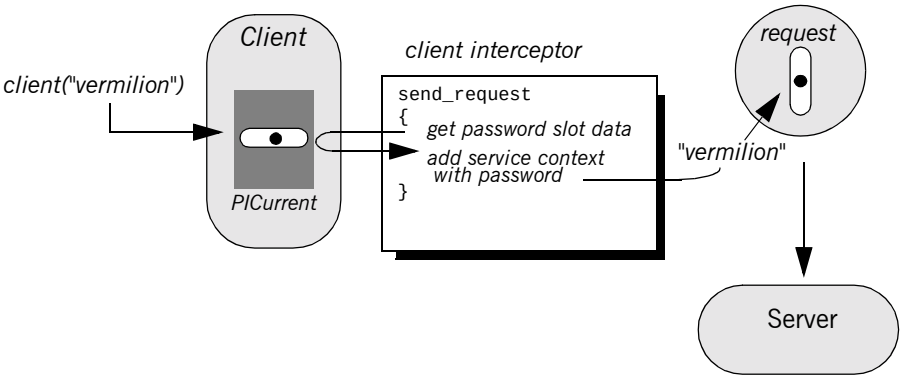

Figure 54: *PICurrent facilitates transfer of thread context data to a request or reply.*

The PortableInterceptor module defines the interface for PICurrent as follows:

```
module PortableInterceptor
{
    // ...
    typedef unsigned long SlotId;
    exception InvalidSlot {};

    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:

```
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 502). 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 515).

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

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:

```
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 "Creating and Registering Policy Factories" on page 532).

ORB Initializer

ORB initializers implement interface `PortableInterceptor::OrbInitializer`:

```
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 gives 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:

```
local interface IORInfo {

    CORBA::Policy
    get_effective_policy(in CORBA::PolicyType type);

    void
    add_ior_component(in IOP::TaggedComponent component);

    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.

```
ACL_IORInterceptorImpl::ACL_IORInterceptorImpl(
    IOP::Codec_ptr codec
) IT_THROW_DECL(()) :
    m_codec(IOP::Codec::_duplicate(codec))
{
}

void
ACL_IORInterceptorImpl::establish_components(
    PortableInterceptor::IORInfo_ptr ior_info
) IT_THROW_DECL((CORBA::SystemException))
{
    CORBA::Boolean requires_password = IT_FALSE;

    try {
1   CORBA::Policy_var policy =
        ior_info->get_effective_policy(
            AccessControl::PASSWORD_POLICY_ID);
    AccessControl::PasswordPolicy_var password_policy =
        AccessControl::PasswordPolicy::_narrow(policy);
    assert(!CORBA::is_nil(password_policy));

2   requires_password = password_policy->requires_password();
    }
    catch (const CORBA::INV_POLICY&) {
    // Policy wasn't set...don't add component
    }
```

```
        CORBA::Any component_data_as_any;
        component_data_as_any <<=
            CORBA::Any::from_boolean(requires_password);

3       CORBA::OctetSeq_var  octets =
            m_codec->encode_value(component_data_as_any);
4       IOP::TaggedComponent component;
        component.tag = AccessControlService::TAG_REQUIRES_PASSWORD;
        component.component_data.replace(octets->length(),
                        octets->length(),
                        octets->get_buffer(),
                        IT_FALSE);

5       ior_info->add_ior_component(component);
    }
```

The sample application's implementation of `establish_components()` executes as follows:

1. Gets the effective password policy object for the POA by calling `get_effective_policy()` on the `IORInfo`.
2. Gets the password policy value by calling `requires_password()` on the policy object.
3. Encodes the password policy value as an octet.
4. Instantiates a tagged component (`IOP::TaggedComponent`) and initializes it with the `TAG_REQUIRES_PASSWORD` tag and encoded password policy value.
5. 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.

RequestInfo Interface

The RequestInfo interface is defined as follows:

```
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 *Orbix 2000 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 31 on page 513 and Table 32 on page 525 show which RequestInfo operations and attributes are valid for a given interception point.

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:

```
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 points implementations can narrow their `ClientRequestInfo` or `ServerRequestInfo` objects to this interface. The two attributes apply to different interception points, as follows:

Table 30:

Timeout attribute	Relevant interception points

`request_end_time`	`send_request`
	`send_poll`
	`receive_request_service_contexts`
	`receive_request`

`reply_end_time`	`send_reply`
	`send_exception`
	`send_other`
	`receive_reply`
	`receive_exception`
	`receive_other`

Writing Client Interceptors

Client interceptors implement the `ClientRequestInterceptor` interface, which defines five interception points:

```
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 514.

As noted earlier, the ORB initializer instantiates and registers the client interceptor. This interceptor’s constructor is implemented as follows:

```
// Client interceptor constructor
ACL_ClientInterceptorImpl::ACL_ClientInterceptorImpl(
    PortableInterceptor::SlotId password_slot,
    IOP::Codec_ptr              codec
) IT_THROW_DECL(()) :
    m_password_slot(password_slot),
    m_codec(IOP::Codec::_duplicate(codec))
{
}
```

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 346)

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 55 depicts a successful reply-request sequence, where the server returns a normal reply:

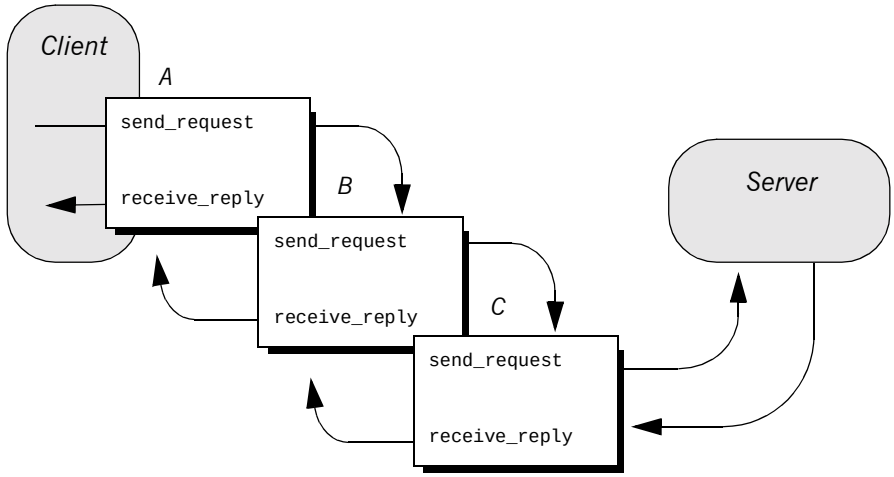

Figure 55: *Client interceptors process a normal reply.*

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 56:

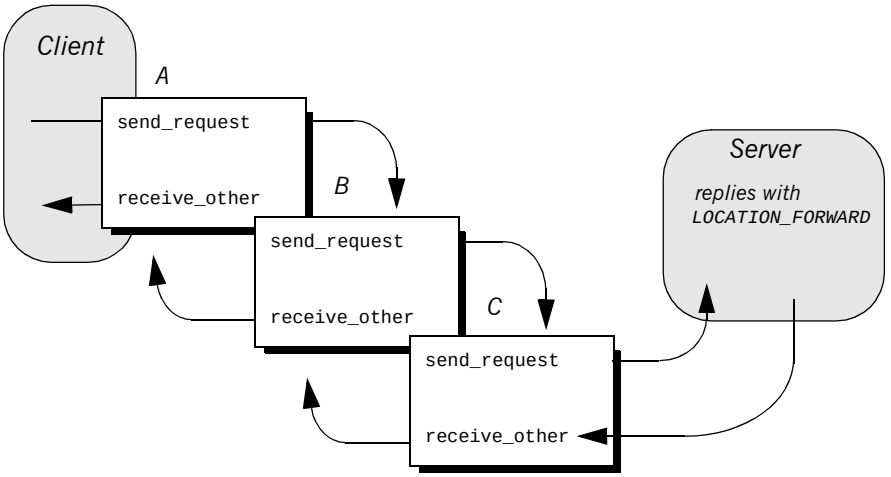

Figure 56: *Client interceptors process a LOCATION_FORWARD reply.*

Scenario 3: Exception aborts interception flow

Any number of events can abort or shorten the interception flow. Figure 57 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.

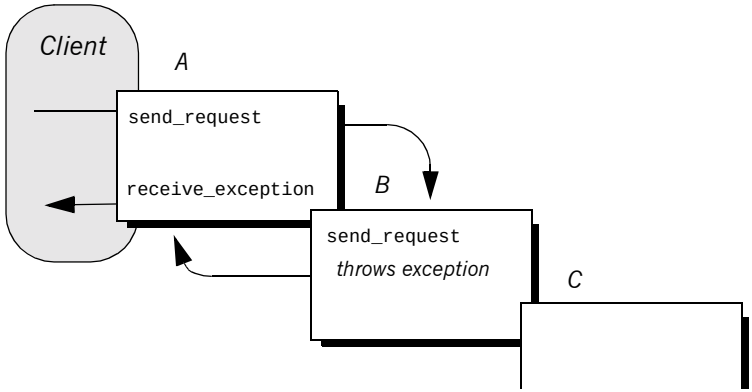

Figure 57: *send_request throws an exception in a client-side interceptor*

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 58 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.

Note: Interceptors must never change the CompletionStatus of the received exception.

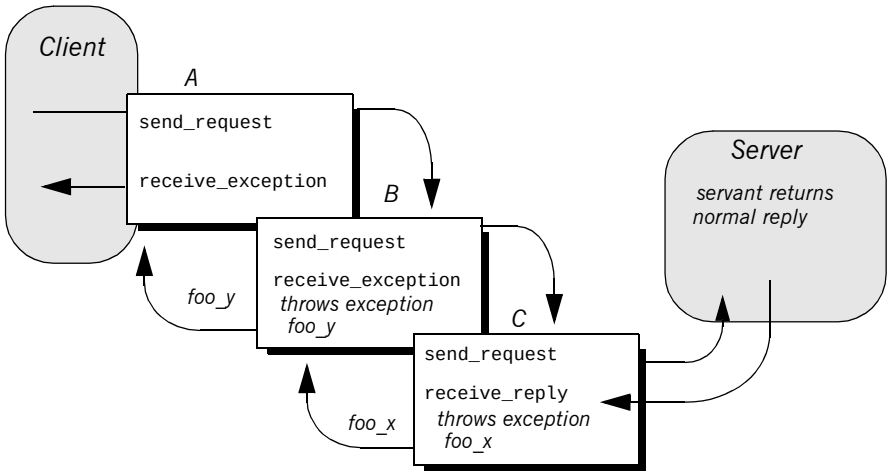

Figure 58: *Client interceptors can change the nature of the reply.*

ClientRequestInfo

Each interception point gets a single `ClientRequestInfo` argument, which provides the necessary hooks to access and modify client request data:

```
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 31 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 31: *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

Table 31: *Client Interception Point Access to ClientRequestInfo*

ClientRequestInfo:	s_req	s_poll	r_reply	r_exep	r_other
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
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 *Orbix 2000 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:

```
void
ACL_ClientInterceptorImpl::send_request(
    PortableInterceptor::ClientRequestInfo_ptr request
) IT_THROW_DECL((
    CORBA::SystemException,
    PortableInterceptor::ForwardRequest
))

try {
// Check if the object requires a password
1     if (requires_password(request))
    { // ...
    }
}

// ...

CORBA::Boolean
ACL_ClientInterceptorImpl::requires_password(
```

```
        PortableInterceptor::ClientRequestInfo_ptr request
    ) IT_THROW_DECL((CORBA::SystemException))
    {
        try {
2       IOP::TaggedComponent_var password_required_component =
            request->get_effective_component(
            AccessControlService::TAG_REQUIRES_PASSWORD
            );

3       IOP::TaggedComponent::_component_data_seq& component_data =
            password_required_component->component_data;
        CORBA::OctetSeq octets(component_data.length(),
                  component_data.length(),
                  component_data.get_buffer(),
                  IT_FALSE);

4       CORBA::Any_var password_required_as_any =
            m_codec->decode_value(octets, CORBA::_tc_boolean);

        CORBA::Boolean password_required;
5       if (password_required_as_any >>=
            CORBA::Any::to_boolean(password_required))
        {
            return password_required;
        }
        }
        catch (const CORBA::BAD_PARAM&)
        {
        // Component does not exist; treat as not requiring a password
        }

        return IT_FALSE;
    }
```

The interception point executes as follows:

1. Calls the subroutine `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. `component_data()` returns the tagged component's data as an octet sequence.

4. 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.
5. The Any is evaluated to determine whether the component data of TAG_REQUIRES_PASSWORD is set to true.

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:

```
// Get the specified password
CORBA::Any_var password =
     request->get_slot(m_password_slot);
// ...
}
```

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:

```
// Encode the password as a service context
CORBA::OctetSeq_var octets =
    m_codec->encode_value(password);
IOP::ServiceContext::_context_data_seq seq(
                            octets->length(),
                            octets->length(),
                            octets->get_buffer(),
                            IT_FALSE);
```

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():

```
IOP::ServiceContext service_context;
service_context.context_id   =
    AccessControlService::PASSWORD_SERVICE_ID;
```

```
service_context.context_data = seq;

request->add_request_service_context(
    service_context, IT_TRUE);
```

Writing Server Interceptors

Server interceptors implement the `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 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 59 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_reply` 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.

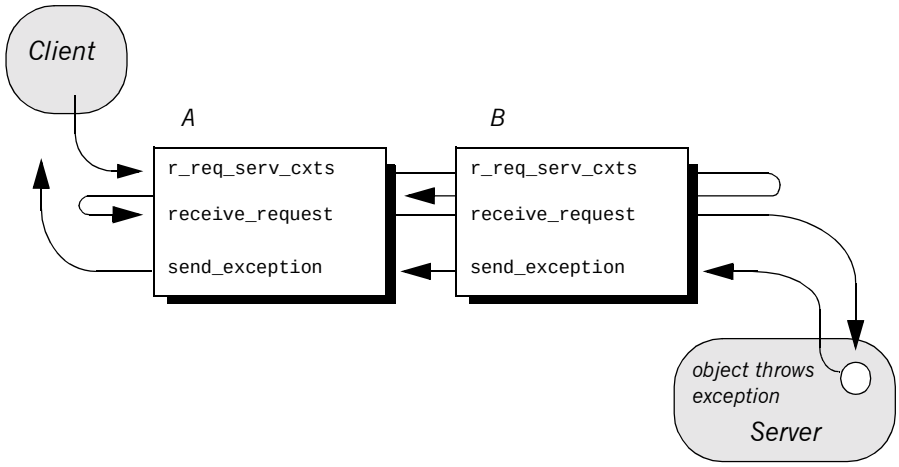

Figure 59: *Server interceptors receive request and send exception thrown by target object.*

Scenario 2: Exception aborts interception flow

Any number of events can abort interception flow. Figure 60 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.

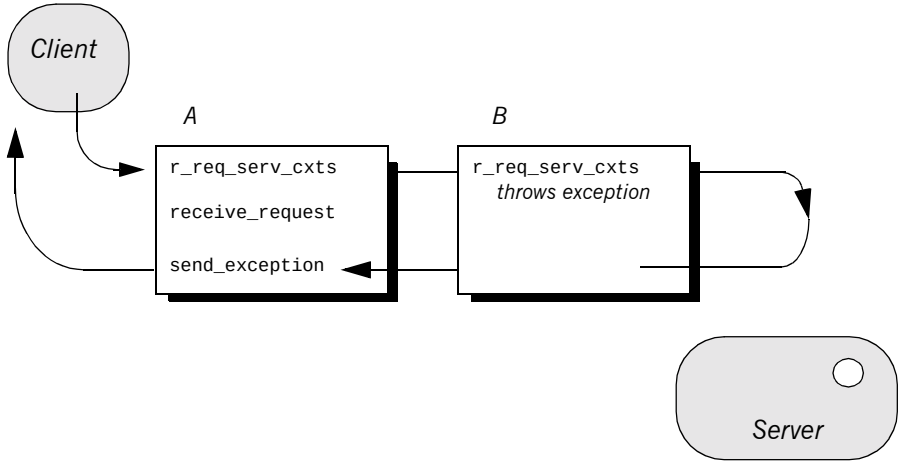

Figure 60: *receive_request_service_contexts throws an exception and interception flow is aborted.*

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 61 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.

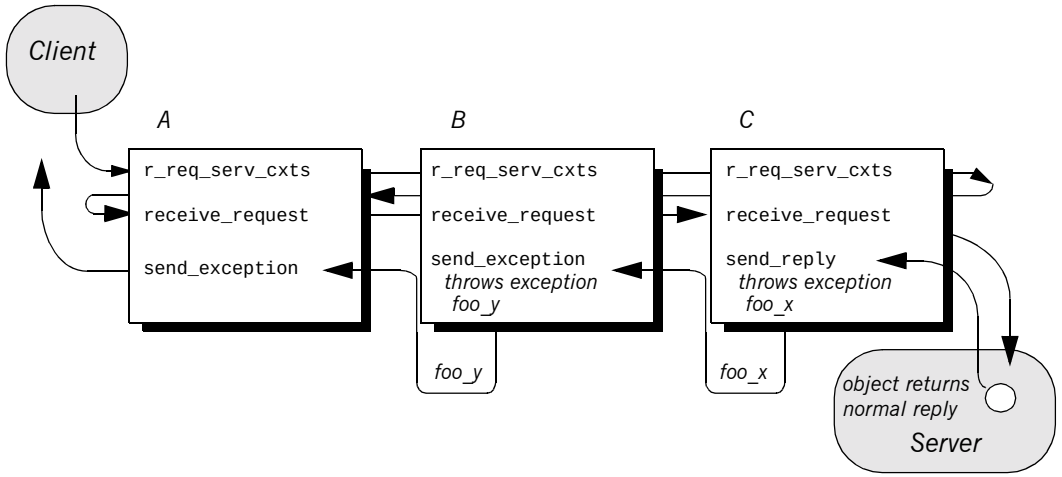

Figure 61: *Server interceptors can change the reply type.*

Note: Interceptors must never change the CompletionStatus of the received exception.

ServerRequestInfo

Each interception point gets a single `ServerRequestInfo` argument, which provides the necessary hooks to access and modify server request data:

```
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 32 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 32: *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	
get_server_policy	y	y	y	y	y
set_slot	y	y	y	y	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 *Orbix 2000 Programmer's Reference*.

Getting 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`:

```
void
ACL_ServerInterceptorImpl::receive_request_service_contexts(
    PortableInterceptor::ServerRequestInfo_ptr request
) IT_THROW_DECL((
    CORBA::SystemException,
    PortableInterceptor::ForwardRequest
))
{
    // Determine whether password protection is required.
    AccessControl::PasswordPolicy_var password_policy =
    get_password_policy(request);
// ...

AccessControl::PasswordPolicy_ptr
ACL_ServerInterceptorImpl::get_password_policy(
    PortableInterceptor::ServerRequestInfo_ptr request
```

```
) IT_THROW_DECL((CORBA::SystemException))
{
    try {
    CORBA::Policy_var policy = request->get_server_policy(
        AccessControl::PASSWORD_POLICY_ID);
    return AccessControl::PasswordPolicy::_narrow(policy);
    }
    catch (const CORBA::INV_POLICY&) {
    // Policy not specified
    }

    return AccessControl::PasswordPolicy::_nil();
}

// ...
```

Getting 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`:

```
    // ...
    if (!CORBA::is_nil(password_policy) &&
    password_policy->requires_password())
    {
        CORBA::String_var server_password =
            password_policy->password();
        if (!check_password(request, server_password))
        {
            throw CORBA::NO_PERMISSION(0xDEADBEEF);
        }
    }
// ...

CORBA::Boolean
ACL_ServerInterceptorImpl::check_password(
    PortableInterceptor::ServerRequestInfo_ptr request,
    const char*                                expected_password
) IT_THROW_DECL((CORBA::SystemException))
{
    try {
```

```
        // Get the password service context...
1       IOP::ServiceContext_var password_service_context =
            request->get_request_service_context(
            AccessControlService::PASSWORD_SERVICE_ID
            );

        // ...convert it into string format...
2       IOP::ServiceContext::_context_data_seq& context_data =
            password_service_context->context_data;
3       CORBA::OctetSeq octets(context_data.length(),
                       context_data.length(),
                       context_data.get_buffer(),
                       IT_FALSE);

4       CORBA::Any_var password_as_any =
            m_codec->decode_value(octets, CORBA::_tc_string);
        const char* password;
        password_as_any >>= password;

        // ...and compare the passwords
5       return (strcmp(password, expected_password) == 0);
        }
        catch (const CORBA::BAD_PARAM&)
        {
        // Service context was not specified
        return IT_FALSE;
        }
    }
```

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. `context_data()` returns the service context data as an octet sequence (see "Service Contexts" on page 497).
3. Initializes an octet sequence with the context data.
4. Calls `decode_value()` on the interceptor's `Codec` to decode the octet sequence into a `CORBA::Any`. The call specifies to extract the string data that is embedded in the octet sequence.

5. 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.

Obtaining 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.

```
void
ACL_ORBInitializerImpl::pre_init(
    PortableInterceptor::ORBInitInfo_ptr  info
) IT_THROW_DECL((CORBA::SystemException))
{
    // Reserve a slot for the password current
1   PortableInterceptor::SlotId password_slot =
        info->allocate_slot_id();

    PortableInterceptor::Current_var pi_current;

    // get PICurrent
    try {
2   CORBA::Object_var init_ref =
        info->resolve_initial_references("PICurrent");
    pi_current = PortableInterceptor::Current::_narrow(init_ref);
    } catch
        (const PortableInterceptor::ORBInitInfo::InvalidName&) {
        throw CORBA::INITIALIZE();
    }
// ...
}
```

Registering 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:

```
// IDL
module AccessControl {
    // ...
    local interface Current : CORBA::Current {
        attribute string password;
    };
};
```

The application defines its implementation of `AccessControl::Current` as follows:

```
#include <omg/PortableInterceptor.hh>
#include <orbix/corba.hh>
#include "access_control.hh"

class ACL_CurrentImpl :
    public AccessControl::Current,
    public IT_CORBA::RefCountedLocalObject
{
  public:
    ACL_CurrentImpl(
    PortableInterceptor::Current_ptr pi_current,
    PortableInterceptor::SlotId      password_slot
    ) IT_THROW_DECL(());

    char*
    password() IT_THROW_DECL((CORBA::SystemException));

    void
    password(const char* the_password
    ) IT_THROW_DECL((CORBA::SystemException));
    // ...
}
```

With `AccessControl::Current` thus defined, the ORB initializer performs these tasks:

1. Instantiates the `AccessControl::Current` object.
2. Registers it as an initial reference.

```
try {
1     AccessControl::Current_var current =
          new ACL_CurrentImpl(pi_current, password_slot);
2     info->register_initial_reference(
      "AccessControlCurrent", current);
}
   catch (const PortableInterceptor::ORBInitInfo::DuplicateName&)
{
    throw CORBA::INITIALIZE();
}
```

Creating and Registering Policy Factories

The sample application's IDL defines the following password policy to provide password protection for the server's POAs.

```
// IDL
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:

```
PortableInterceptor::PolicyFactory_var passwd_policy_factory =
    new ACL_PasswordPolicyFactoryImpl();
info->register_policy_factory(
    AccessControl::PASSWORD_POLICY_ID,
    passwd_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.

Creating 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`:

```
IOP::CodecFactory_var codec_factory = info->codec_factory();
IOP::Encoding cdr_encoding  = { IOP::ENCODING_CDR_ENCAPS, 1, 2 };
IOP::Codec_var cdr_codec =
    codec_factory->create_codec(cdr_encoding);
```

When the ORB initializer instantiates portable interceptors, it supplies this Codec to the interceptor constructors.

Registering 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 534)

The ORB initializer instantiates and registers these interceptors as follows:

```
// Register IOR interceptor
PortableInterceptor::IORInterceptor_var ior_icp =
    new ACL_IORInterceptorImpl(cdr_codec);
info->add_ior_interceptor(ior_icp);

// Register client interceptor
PortableInterceptor::ClientRequestInterceptor_var client_icp =
new ACL_ClientInterceptorImpl(password_slot, cdr_codec);
info->add_client_request_interceptor(client_icp);

// Register server interceptor
PortableInterceptor::ServerRequestInterceptor_var server_icp =
new ACL_ServerInterceptorImpl(cdr_codec);
info->add_server_request_interceptor(server_icp);
```

Registering an ORBInitializer

An application registers an ORB initializer by calling `register_orb_initializer`, which is defined in the `PortableInterceptor` name space as follows:

```
namespace PortableInterceptor {
    static void register_orb_initializer(
        PortableInterceptor::ORBInitializer_ptr init);
};
```

Each service that implements interceptors provides an instance of an ORB initializer. To use a service, an application follows these steps:

1. Calls `register_orb_initializer` and supplies the service's ORB initializer.
2. Instantiates a new ORB by calling `ORB_init()` with a new ORB identifier.

An ORB initializer is called by all new ORBs that are instantiated after its registration.

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 *Orbix 2000 Administrator's Guide*.

Appendix A
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 plugin switch, such as `-poa` or `-base`, unless you modify the IDL configuration file to set `IsDefault` for one or more plugins to Yes. (see page 544). As distributed, the configuration file sets `IsDefault` for all plugins 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
`-D`*`name[:value]`*	Defines the preprocessor's name.
`-E`	Runs preprocessor only, prints on `stdout`.
`-I`*`dir`*	Includes *dir* in search path for preprocessor.
`-R`*`[-v]`*	Populates the interface repository (IFR). The -v modifier specifies verbose mode.
`-U`*`name`*	Undefines name for preprocessor.

Switch	Description
`-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 *plugin* to generate code that is specific to a language or ART plug-in. You must specify at least one plugin to the idl compiler Use one of these values for *plugin*: • `base`: Generate C++ header and stub code. • `jbase`: Generate Java stub code • `poa`: Generate POA code for C++ servers. • `jpoa`: 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 *plugin* 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 33: *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.

`-i`*`path-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 33: *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 36).

`-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 34: *Modifier for -base, -psdl, and -pss_r plugin 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`

-xOBV	Generates object-by-value default `valuetype` implementations in files.

Table 35: *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.

Table 35: *Modifiers for -jbase and -jpoa switches*

Modifier	Description
-VTRUE -VFALSE	Generate native implementation for valuetypes. The default is FALSE.
-FTRUE -FFALSE	Generate factory implementation for valuetypes. The default is FALSE.

Table 36: *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`

Table 36: *Modifiers for -poa switch*

Modifier	Description

-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 plugin 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"`

-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` plugin 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:

```
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:

```
# 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";
};

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";
};

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 Foundation Classes Library

For each platform, IONA distributes several variants of its IONA foundation classes (IFC) shared library, which provides a number of proprietary features, such as a threading abstraction. For each IFC library, IONA provides checked and unchecked variants:

- Checked variants are suitable for development and testing: extra checking is built into the code—for example, it throws an exception when a thread attempts to lock a mutex that it has already locked.
- Unchecked variants are suitable for deployed applications, which have been tested for thread safety.

Each UNIX distribution provides IFC libraries that support the POSIX thread package. The following platforms have multiple IFC libraries, which support different thread packages:

Platform	Thread package support
HPUX 32	POSIX, DCE/CMA
Solaris 32/64	POSIX, UI

Installed IFC Directories

Each Orbix installation makes IFC variants available in directories with this format:

Unix	
Checked	`$IT_PRODUCT_DIR/shlib/`*`native-thread-pkg`*`/libit_ifc_`*`compiler-spec`*
Unchecked	`$IT_PRODUCT_DIR/shlib/`*`native-thread-pkg`*`/checked/libit_ifc_`*`compiler-spec`*
Windows	
Checked	`%IT_PRODUCT_DIR%\bin\windows\it_ifc3_vc60.dll`
Unchecked	`%IT_PRODUCT_DIR%\bin\windows\checked\it_ifc3_vc60.dll`

Further, each installation provides a default IFC directory, which contains an unchecked variant. On UNIX platforms, the default directory contains a symbolic link to an unchecked variant of UI or POSIX; on Windows, it contains a copy of the unchecked variant of the Windows IFC library:

UNIX:

`$IT_PRODUCT_DIR/shlib/default/`*`ifc-lib-sym-link`*

Windows:

`%IT_PRODUCT_DIR%\bin\it_ifc3_vc60.dll`

Selecting an IFC Library

Options for setting a given program's IFC library are platform-dependent.

Unix

On UNIX systems, you can set a program's IFC library in two ways:

- (Recommended) When linking the program, use the linker's run path feature, and set it to the desired IFC library directory. For example, set the `-R` option with the Sun compiler.
- Set the program's environment variable (`LD_LIBRARY_PATH` or `SHLIB_PATH`). Keep in mind that other services such as the Locator also might use this environment and can be affected by this setting.

Windows

Set `PATH` to the desired IFC library directory.

Index

C

D

H

I

O

P

Q

R

S

T

U

V

W