OrbixWeb
Programmer's Guide

IONA Technologies PLC
September 1998

IONA Technologies PLC
The IONA Building
Shelbourne Road
Dublin 4
Ireland
Phone: . . . . . +353-1-662 5255
Fax: . . . . . +353-1-662 5244

IONA Technologies Inc.
60 Aberdeen Ave.
Cambridge, MA 02138
USA
Phone: . . . . . +1-617-949-9000
Fax: . . . . . +1-617-949-9001

IONA Technologies Pty. Ltd.
Ashton Chambers, Floor 3
189 St. George's Terrace
Perth WA 6000
Australia
Phone: . . . . . +61 9 288 4000
Fax: . . . . . +61 9 288 4001

Support: . . . . . support@iona.com
Training: . . . . . training@iona.com
Orbix Sales: . . . . . sales@iona.com
IONA's FTP site . . . . . ftp.iona.com
World Wide Web: . . . . . http://www.iona.com/

OrbixWeb is a Registered Trademark of IONA Technologies PLC.

COPYRIGHT NOTICE

M 2 1 4 7

Contents

OrbixWeb Programmer's Guide

Part 1 Getting Started

Part II CORBA Programming with OrbixWeb

Part II Running OrbixWeb Programs

Part IV Topics in OrbixWeb Programming

Part V Advanced CORBA Programming

Part VI Advanced OrbixWeb Programming

Preface

OrbixWeb 3.1 is an implementation of the Common Object Request Broker Architecture (CORBA) from the Object Management Group (OMG) that maps CORBA functionality to the Java programming language. OrbixWeb combines a powerful standards-based approach to distributed application development with the flexibility and ease of use of the Java environment.

OrbixWeb 3.1 is available in two editions:

- OrbixWeb 3.1 Standard Edition.
- OrbixWeb 3.1 Professional Edition.

  The Professional Edition of OrbixWeb 3.1 augments the Standard development kit with full Naming Service, Interface Repository and Server Manager utilities.

  OrbixWeb 3.1 Professional Edition also ships with Wonderwall, IONA's firewall for Internet Inter-ORB Protocol (IIOP) communication.

Audience

The *OrbixWeb Programmer's Guide* and the *OrbixWeb Programmer's Reference* are intended for use by application programmers and designers wishing to familiarise themselves with CORBA distributed programming and its application in the Java environment. These guides assume as a prerequisite that you are familiar with the Java programming language.

Overview of OrbixWeb 3.1

The Internet ORB

OrbixWeb 3.1 features tight integration with Wonderwall ensuring that OrbixWeb applications are fully Internet-enabled. This includes automatic runtime support for IOR firewall profiles and IIOP options to allow for Internet callback behaviour. The OrbixWeb

runtime has also been upgraded allowing OrbixWeb applications to be the fully SSL V3 enabled.

The Java Server

OrbixWeb 3.1 includes a Java version of the server activation component (`orbixd`) and associated utilities. The pure Java Daemon (`orbixdj`) allows you to launch Java CORBA server components either as new threads within the VM running `orbixdj` or alternatively, in separately launched Java VMs. Using server threads within a single VM presents a natural and scalable approach for Java server deployment. You can also avail of the full OrbixWeb server bind and auto-activation capability on any Java-enabled platform with file-system support.

Several features of `orbixd` that were not supported by `orbixdj` in OrbixWeb 3.0 have now been added to OrbixWeb 3.1. These include invoke and launch rights and locator functionality.

Ease of Use

In OrbixWeb 3.1 all configuration information is stored in a central (downloadable) properties file and all configuration is performed using Java-based GUI components. No command-line editing of configuration data is required. Graphical user interfaces are also provided for Server Manager, IFR and Naming Service utilities; while demo-suite and documentation are extended to make OrbixWeb as simple as possible to use 'out-of-the-box'. OrbixWeb 3.1 ships with a new Server Manager GUI.

IDL/Java Mapping

OrbixWeb provides full and complete support for the standard OMG IDL to Java Mapping (version 1.1) including the Java ORB portability interfaces.

New Features

Multiple ORB Support

OrbixWeb 3.1 provides support for multiple ORBs. This gives applications enhanced flexibility as each newly-created ORB is completely independent from any other ORB; for example, in its configuration settings, connections, and listener ports. This facilitates complete applet separation in browsers.

Enhanced Diagnostics and Instrumentation Support

The OrbixWeb runtime now produces full diagnostics output to aid you with debugging. Diagnostics are broken down by component, each associated with a particular diagnostics level. The basic instrumentation support offered by OrbixWeb enables you to log instrumentation data for specified events.

IIOP 1.1 Fragmentation

OrbixWeb 3.1 complies with version 1.1 of the Internet Inter-ORB Interoperability Protocol (IIOP). OrbixWeb now allows you to send IIOP messages as fragments. This increases parallelism and improves the overall dispatch speed for very large messages. It also brings the additional benefit of lower memory consumption.

Service Contexts

OrbixWeb 3.1 includes support for service contexts. The OrbixWeb service context API allows you to pass service specific information in IIOP message headers. You can use service contexts on a per-object or per-request basis.

Java Daemon Graphical Console

The Java Daemon graphical console has also been extended to produce full diagnostics output. In addition, you can now use the Java Daemon console to output threading information and to run garbage collection.

Organisation of the OrbixWeb Documentation

This section gives a summary of the structure of the *OrbixWeb Programmer's Guide*, and a brief outline of the *OrbixWeb Programmer's Reference*.

The OrbixWeb Programmer's Guide

The *OrbixWeb Programmer's Guide* is divided into six parts. Parts I, II and III provide the basis for understanding the remainder of the material covered in these guides.

Part I Getting Started

This part of the guide introduces basic CORBA concepts, and introduces OrbixWeb by describing a simple programming example. It works through the steps required to write client and server Java applications. This also provides an example of integrating client functionality with Java applets.

Many of the concepts that form the basis of Part II are introduced in this part.

Part II CORBA Programming with Orbix Web

Part II provides a more complete description of developing CORBA programs in Java using OrbixWeb.

This part of the guide provides an outline of the CORBA Interface Definition language (IDL) and the standard OMG mapping from IDL to Java. It shows how to program a simple application and provides information on various aspects of programming a distributed application, including the use of the Naming Service to identify objects in the system.

Part III Running OrbixWeb Programs

This part describes the issues involved in running OrbixWeb programs. An important aspect of this description is a complete introduction to the OrbixWeb *Implementation Repository*. The Java Daemon, `orbixdj`, is also introduced.

Part IV Topics in OrbixWeb Programming

This part describes a small set of miscellaneous features, most of which are commonly used in OrbixWeb applications. These features include the use of exception handling in a distributed system.

Part V Advanced CORBA Programming

This part of the guide explains more advanced features of OrbixWeb as specified by the CORBA standard. In particular, it provides the information needed to use the *Dynamic Invocation Interface* that allows a client to issue requests on objects whose interfaces may not have been defined at the time the application was compiled.

Part VI Advanced OrbixWeb Programming

OrbixWeb provides a number of interfaces to allow you to influence runtime behaviour for particular deployment scenarios. Part VI explains how you can replace different components of OrbixWeb, and the circumstances where the use of these OrbixWeb specific features is advantageous.

The OrbixWeb Programmer's Reference

The *OrbixWeb Programmer's Reference* expands on some information presented in the *OrbixWeb Programmer's Guide*, and provides details of the public Application Programming Interfaces (API) in the standard `org.omg.CORBA` package, as well as the details of the OrbixWeb API. The *OrbixWeb Programmer's Reference* also describes the new GUI tools available in OrbixWeb 3.

A Message from the OrbixWeb Team

In developing Version 3.1, our focus was to provide Java ORB technology which is closest to the spirit of Java and which is completely at home on the Internet. As Java extends from the browser to the middle-tiers and back-end, a Java ORB capable of scaling from the thin client to the enterprise server is required. We have built OrbixWeb 3.1 to be that ORB while keeping in mind that CORBA is a development tool and ease of development, integration and deployment remain paramount.

Document Conventions

This guide uses the following typographical conventions:

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

This guide may use the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, no prompt is used.
%	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
#	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
>	The notation > represents the DOS, Windows NT, or Windows 95 command prompt.
... ⋮	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.

[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices enclosed in { } (braces) in format and syntax descriptions.

Online Support for OrbixWeb

The following resources are available to users seeking further information on OrbixWeb topics:

Documentation Updates

Online documentation for the *OrbixWeb Programmer's Guide* and the *OrbixWeb Programmer's Reference* is updated on a regular basis. You can access this information from the *OrbixWeb Product Pages* on the IONA Web site at:

```
http://www.iona.com.
```

Additional Online Technical Information

The IONA *Knowledge Base* provides detailed and frequently updated technical information on a wide range of OrbixWeb topics.

Access the *Knowledge Base* through the *OrbixWeb Product Pages* on IONA's web site.

Contacting Technical Support at IONA

If you have a support contract with IONA, you can send an e-mail to `support@iona.com`.

Note: You *must* quote your customer registration details in order to have your request processed.

You will receive by return, an automatic e-mail assigning you a Problem Report number. Please quote this PR number in subsequent communications with the IONA Support team.

When submitting requests to IONA Support, please include the following information:

- Your full name and e-mail address (and customer registration details).
- Which platforms and compilers you are using (including version details).
- A detailed description of the problem.

If you do not have a support contract with IONA, and wish to receive information about purchasing support, mail `info@iona.com` to request further details.

Part 1

Getting Started

1

Introduction to CORBA and OrbixWeb

OrbixWeb is a software environment that allows you to build and integrate distributed applications. OrbixWeb is a full implementation of the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) specification. This chapter introduces CORBA and describes how OrbixWeb implements this specification.

CORBA and Distributed Object Programming

The diversity of modern networks makes the task of network programming very difficult. Distributed applications often consist of several communicating programs written in different programming languages and running on different operating systems. Network programmers must consider all of these factors when developing applications.

The Common Object Request Broker Architecture (CORBA) defines a framework for developing object-oriented, distributed applications. This architecture makes network programming much easier by allowing you to create distributed applications that interact as though they were implemented in a single programming language on one computer.

CORBA also brings the advantages of object-oriented techniques to a distributed environment. It allows you to design a distributed application as a set of cooperating objects and to reuse existing objects in new applications.

The Role of an Object Request Broker

CORBA defines a standard architecture for Object Request Brokers (ORBs). 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 role of the ORB is to hide the underlying complexity of network communications from the programmer.

An ORB allows you to create standard software objects whose methods 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*.

When a client invokes a member method on a CORBA object, the ORB intercepts the method call. As shown in Figure 1, the ORB redirects the method call across the network to the target object. The ORB then collects results from the method call and returns these to the client.

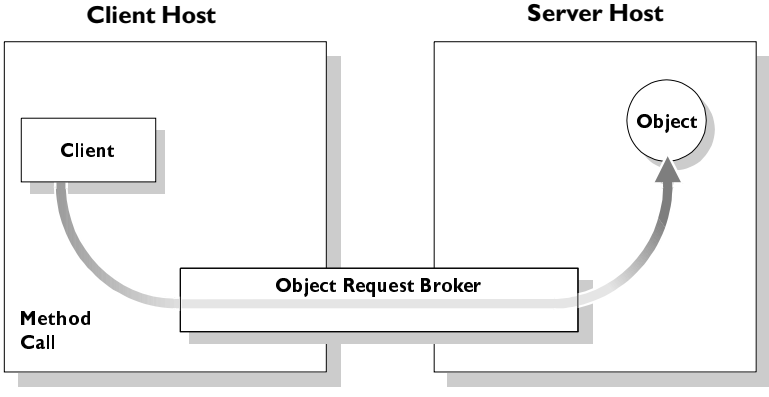

Figure 1: *The Object Request Broker*

The Nature of Objects in CORBA

CORBA objects are standard software objects implemented in any supported programming language. CORBA supports several languages, including Java, C++ and Smalltalk.

With a few calls to an ORB's application programming interface (API), you can make CORBA objects available to client programs in your network. Clients can be written in any supported programming language and can invoke the member methods of a CORBA object using the normal programming language syntax.

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 what member methods are available to a client, without making any assumptions about the implementation of the object.

To invoke member methods on a CORBA object, a client needs only the object's IDL definition. The client does not need to know details such as the programming language used to implement the object, the location of the object in the network, or the operating system on which the object runs.

The separation between an object's interface and its implementation has several advantages. For example, it allows you to change the programming language in which an object is implemented without changing clients that access the object. It also allows you to make existing objects available across a network.

The Structure of a CORBA Application

The first step in developing a CORBA application is to define the interfaces to objects in your system, using CORBA IDL. You then compile these interfaces using an IDL compiler.

An IDL compiler generates Java from IDL definitions. This Java includes *client stub code,* which allows you to develop client programs, and *server skeleton code*, which allows you to implement CORBA objects.

As shown in Figure 2 on page 6, when a client calls a member method on a CORBA object, the call is transferred through the client stub code to the ORB. If the client has not accessed the object before, the ORB refers to a database, known as the *Implementation Repository*, to determine exactly which object should receive the method call. The ORB then passes the method call through the server skeleton code to the target object.

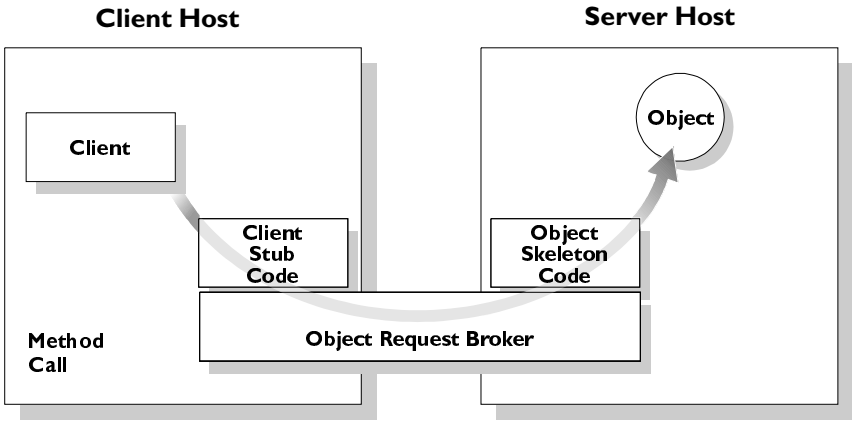

Figure 2: *Invoking on a CORBA Object*

The Structure of a Dynamic CORBA Application

One difficulty with normal CORBA programming is that you have to compile the IDL associated with your objects and use the generated Java code in your applications. This means that your client programs can only invoke member methods on objects whose interfaces are known at compile-time. If a client wishes to obtain information about an object's IDL interface at runtime, it needs an alternative, *dynamic* approach to CORBA programming.

The CORBA *Interface Repository* is a database that stores information about the IDL interfaces implemented by objects in your network. A client program can query this database at runtime to get information about those interfaces. The client can then call member methods on objects using a component of the ORB called the *Dynamic call Interface* (DII), as shown in Figure 3 on page 7.

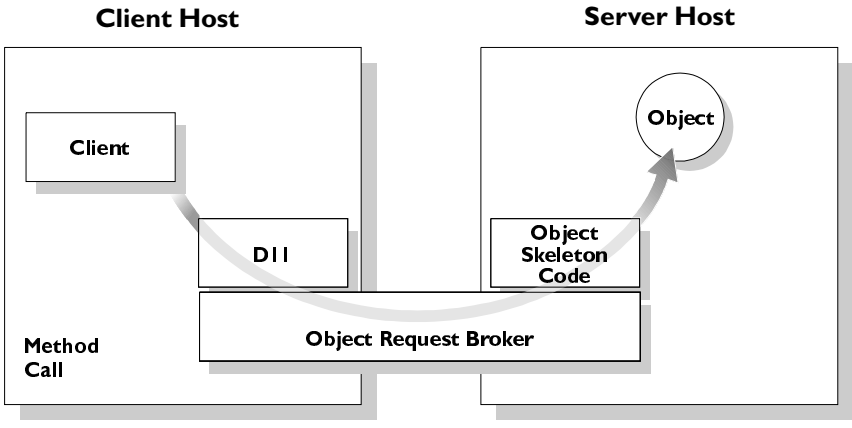

Figure 3: *Client Invoking a Method Using the DII*

CORBA also supports *dynamic* server programming. A CORBA program can receive method calls through IDL interfaces for which no CORBA object exists. Using an ORB component called the *Dynamic Skeleton Interface* (DSI), the server can then examine the structure of these method calls and implement them at runtime. Figure 4 on page 8 shows a dynamic client program communicating with a dynamic server implementation.

Note: The implementation of Java interfaces in client-side generated code supplies proxy functionality to client applications. This must not be confused with the implementation of *IDL interfaces* in OrbixWeb servers.

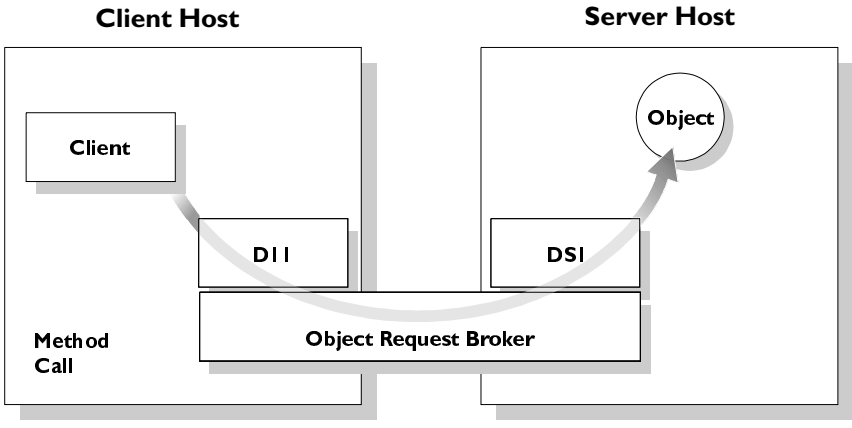

Figure 4: *Method Call Using the DII and DSI*

Interoperability between Object Request Brokers

The components of an ORB make the distribution of programs transparent to network programmers. To achieve this, the ORB components must communicate with each other across the network.

In many networks, several ORB implementations coexist and programs developed with one ORB implementation must communicate with those developed with another. To ensure that this happens, CORBA specifies that ORB components must communicate using a standard network protocol called the *Internet Inter-ORB Protocol* (IIOP).

The Object Management Architecture

An ORB is one component of the OMG's Object Management Architecture (OMA). This architecture defines a framework for communications between distributed objects. As shown in Figure 5, the OMA includes four elements:

- Application objects.
- The ORB.
- The *CORBAservices*.
- The *CORBAfacilities*.

Application objects are objects that implement programmer-defined IDL interfaces. These objects communicate with each other, and with the CORBAservices and CORBAfacilities, through the ORB. The CORBAservices and CORBAfacilities are sets of objects that implement IDL interfaces defined by CORBA and provide useful services for some distributed applications.

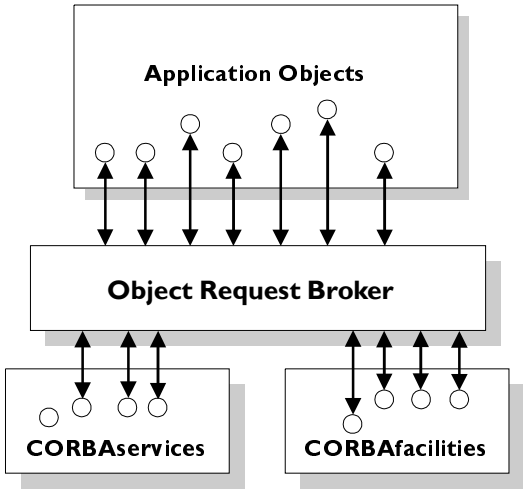

Figure 5: *The Object Management Architecture*

When writing OrbixWeb applications, you may require one or more CORBAservices or CORBAfacilities. This section provides a brief overview of these components of the OMA.

The CORBAservices

The CORBAservices define a set of low-level services that allow application objects to communicate in a standard way. These services include the following:

- The *Naming Service*. Before using a CORBA object, a client program must get an identifier for the object, known as an *object reference*. This service allows a client to locate object references based on abstract, programmer-defined object names.
- The *Trading Service*. This service allows a client to locate object references based on the desired properties of an object.
- The *Object Transaction Service*. This service allows CORBA programs to interact using transactional processing models.
- The *Security Service*. This service allows CORBA programs to interact using secure communications.
- The *Event Service*. This service allows objects to communicate using decoupled, event-based semantics, instead of the basic CORBA function-call semantics.

IONA Technologies implements several CORBAservices including all the services listed above.

The CORBAfacilities

The CORBAfacilities define a set of high-level services that applications frequently require when manipulating distributed objects. The CORBAfacilities are divided into two categories:

- The *horizontal* CORBAfacilities.
- The *vertical* CORBAfacilities.

The horizontal CORBAfacilities consist of user interface, information management, systems management, and task management facilities. The vertical CORBAfacilities standardize IDL specifications for market sectors such as healthcare and telecommunications.

How OrbixWeb Implements CORBA

OrbixWeb is an ORB that fully implements the CORBA 2.0 specification. By default, all OrbixWeb components and applications communicate using the CORBA standard IIOP protocol.

The components of OrbixWeb are as follows:

- The *IDL compiler* parses IDL definitions and produces Java code that allows you to develop client and server programs.
- The *OrbixWeb runtime* is called by every OrbixWeb program and implements several components of the ORB, including the DII, the DSI, and the core ORB functionality.
- The *OrbixWeb daemon* is a process that runs on each server host and implements several ORB components, including the Implementation Repository. An all-Java counterpart to the daemon process is also included. This daemon process is known as the Java Daemon, also referred to as `orbixdj`.
- The *OrbixWeb Interface Repository server* is a process that implements the Interface Repository.

OrbixWeb also includes several programming features that extend the capabilities of the ORB. These features are described in the Advanced OrbixWeb Programming section of this guide.

The *OrbixWeb GUI Tools* and the *OrbixWeb command-line utilities* allow you to manage and configure the components of OrbixWeb.

2

Getting Started with Java Applications

This chapter introduces OrbixWeb with a step by step description of how to create a simple grid application. These steps include defining an Interface Definition Language (IDL) interface, implementing this interface in Java, and developing a standalone client application. The OrbixWeb IDL Compiler and the files it generates are also introduced in this chapter.

The example application used is a two-dimensional grid, implemented by a Java class in a server. The grid example illustrates how Java applications can act as clients to servers containing objects that implement IDL definitions. The server and client can run on different machines in a distributed system.

The grid example is used because it gives an abstract view of commonly-used components, such as spreadsheets or relational tables. The sample code described in this chapter is available in the `demos/grid` directory of your OrbixWeb installation.

OrbixWeb Programming Steps

The following programming steps are required to create a distributed client/server application in Java using OrbixWeb:

1. Defining the IDL interface.
2. Compiling the IDL interface.
3. Implementing the IDL interface.
4. Writing the server application.
5. Writing the client application.
6. Compiling the client and server.
7. Registering the server.
8. Running the client.

This section outlines these programming steps in detail, using the grid demonstration as an example.

Defining the IDL Interface

The first step in writing an OrbixWeb program is to define the interfaces to the application objects, using IDL.

The interface to the grid application can be defined in IDL as follows:

```
// IDL
// gridDemo.idl
interface grid {
   readonly attribute short height;
   readonly attribute short width;

   void set(in short row, in short col, in long value);
   long get(in short row, in short col);
};
```

This IDL interface has two *attributes* (`height` and `width`) that define the number of rows and columns in the grid. The interface has two *operations* corresponding to methods that clients can call on the object:

`set()` This operation sets the grid element `[rol,col]` to `value`.

`get()`	This operation returns grid element `[row,col]`.

The parameters to these operations are labelled as `in`. This means that the parameters are passed from the client to the server. In other interfaces, parameters may be labelled as `out` (from the server to the client) or `inout` (in both directions).

Compiling the IDL Interface

The next programming step is to compile the IDL interface using the IDL compiler. This checks the validity of the IDL specification, and generates Java code.

Setting Up the Configuration File for the IDL Compiler

Before running the IDL compiler you should check that OrbixWeb can find the configuration file, `Orbix.cfg`.

Windows

If OrbixWeb is installed on Windows, the environment variable `IT_CONFIG_PATH` must be set to point to the directory in which `Orbix.cfg` resides; typically, this is the installation directory. If `IT_CONFIG_PATH` is not set, the following registry entry is used:

```
HKEY_LOCAL_MACHINE\SOFTWARE\IONATechnologies\
   OrbixWeb\<Version>\Configuration\IT_CONFIG_PATH
```

UNIX

Set the environment variable `IT_CONFIG_PATH` to point to the directory in which `Orbix.cfg` resides, using one of the following scripts:

```
setenvs.sh
setenvs.csh
```

If the environment variable `IT_CONFIG_PATH` is not set, the compiler looks for `Orbix.cfg` in the directory `/etc`.

Running the IDL Compiler

To compile the IDL interface, enter the following command at the operating system prompt:

```
idl -jP gridDemo gridDemo.idl
```

This command generates a number of java files which are used to communicate with OrbixWeb. The generated files are located in the `grid/java_output/gridDemo` directory. Discussion of these files is deferred until the section, "OrbixWeb IDL Compilation" on page 29.
The `jP` switch passed to the IDL compiler specifies the package name into which all generated Java classes are placed. This helps to avoid potential name clashes. In the grid example, all application files are placed within a package called `gridDemo`.

Implementing the IDL Interface

The next step involves implementing the IDL interface using the code generated by the IDL compiler. In the grid example, the *ImplBase* approach is used to implement the IDL interface. The *TIE* approach can also be used. Both of these approaches are discussed in detail in "Implementing the Interfaces" on page 138. Implementing the `grid` IDL interface using the ImplBase approach involves creating an implementation class which derives from the IDL generated class `_gridImplBase`.

In this example, the implementation class created is `GridImplementation`. This class implements the attributes and operations defined in the file `gridDemo.idl`.

```
package gridDemo;

// import OrbixWeb related classes
import IE.Iona.OrbixWeb._OrbixWeb;
import IE.Iona.OrbixWeb._CORBA;
import org.omg.CORBA.ORB;

public class GridImplementation
   extends _gridImplBase {

   // store the height
   short m_height;
   // store the width
   short m_width;
```

```
// a 2D array to hold the grid data
int m_array[][];

public GridImplementation(short height,short width){
   //allocate 2D array
   m_array = new int[height][width];
   //set up height
   m_height = height;
   //set up width
   m_width = width;
}

// implementation of the method that reads
// the height attribute
public short height() {
   return m_height;
}

// implementation of the method that reads
// the width attribute
public short width(){
   return m_width;
}

// implementation of the set operation
public void set(short x, short y, int value) {
   System.out.println
      (“In grid.set () x = “ + x);
   System.out.println
      (“In grid.set () y = “ + y);
   System.out.println
      (“In grid.get () value = “ + value);
   m_array[y][x] = value;
}

// implementation of the get operation
public int get (short x, short y) {
   int value = m_array[x][y];
   System.out.println
      (“In grid.get () x = “ + x);
   System.out.println
      (“In grid.get () y = “ + y);
```

```
        System.out.println
           ("In grid.get () value = " + value);
        return value;
     }
  }
```

Writing the Server Application

The next programming step involves writing a server which creates a CORBA object and initializing the ORB to receive calls. In the grid example, a server application is required to service client requests on the `grid` interface. This server creates an instance of the `GridImplementation` class from "Implementing the IDL Interface" on page 16.

ORB initialization

Because OrbixWeb uses the standard OMG IDL to Java mapping, all clients and servers must call `org.omg.CORBA.ORB.init()` to initialize the ORB. This returns a reference to the `ORB` object. The ORB methods defined by the standard can then be invoked on this instance.

You should use the parmeterized version of the `init()` method, defined as follows:

```
static public org.omg.CORBA.ORB init
                (String[] args, ava.util.Properties props)
```

This method is passed an array of strings, which are command arguments, and a list of Java properties. Either of these values may be null. This version of the `init()` method returns a new fully functional ORB Java object each time it is called.

Note: Calling `ORB.init()` without parameters returns a singleton ORB with restricted functionality.

Refer to the *OrbixWeb Programmer's Reference* for further details on the `org.omg.CORBA.ORB` class.

The following server program creates a `GridImplementation` object, and gives it an initial size. It then indicates to OrbixWeb that the server initialization is complete by a call to `impl_is_ready()`. This call takes as a parameter the name of the server as registered with the OrbixWeb daemon.

```
package gridDemo;

import IE.Iona.OrbixWeb._OrbixWeb;
import IE.Iona.OrbixWeb._CORBA;
import org.omg.CORBA.ORB;
import org.omg.CORBA.SystemException;

public class javaserver1 {
   public static void main (String args[]) {
      // constants used to define the size of the grid
      final short width = 100;
      final short height = 100;
1     grid gridImpl = null;

2     ORB orb = ORB.init (args,null);

      try {
3        gridImpl = new GridImplementation (width,height);

4        _CORBA.Orbix.impl_is_ready ("gridServer");

         System.out.println ("Shutting down gridServer...");
5        orb.shutdown (gridImpl);
6     }
7     catch (SystemException se) {
         System.out.println ("Exception during creation of
                              GridImplementation" + se.toString());
         System.exit(1);
      }
      System.out.println ("Grid Server exiting....");
   }
}
```

This code is described as follows:

1. Create a proxy grid to hold the `GridImplementation` object.
2. Initialize the ORB for an OrbixWeb application.
3. Create an implementation of the `grid` server object giving it an initial size of 100 by 100.
4. The call to `impl_is_ready` indicates to OrbixWeb that the server has been initialized and is ready to receive requests on its objects. In this

case, it receives requests on the `gridImpl` object. This is a blocking call which returns when the server times out.

The `impl_is_ready()` call takes the name of the server as registered with the OrbixWeb daemon as a parameter. The `_CORBA.Orbix` object which calls `impl_is_ready()` is used to communicate directly with OrbixWeb.

5. Indicate to OrbixWeb that all calls are finished and finalize the ORB.
6. If the `impl_is_ready()` call should fail the exception will be caught by the `catch` block.

Note: You must enclose all remote operations in a `try..catch` block, otherwise the code will not compile.

Refer to "Implementing the Interfaces" on page 138 for more details on using the `impl_is_ready()` method.

Error Handling for Server Applications

If an error occurs during an OrbixWeb method call, the method may raise a Java exception to indicate this. To handle these exceptions, you must enclose Orbixweb calls in `try` statements. Exceptions thrown by OrbixWeb calls can then be handled by subsequent Java `catch` clauses. All OrbixWeb system exceptions inherit from the class `org.omg.CORBA.SystemException`.

In the grid example, the code in the `catch` clause displays details of possible system exceptions raised by OrbixWeb. It achieves this by printing the result of the `SystemException.toString()` method to the Java `System.out` print stream. The constructor for the IDL generated `_gridImplBase` type may raise a system exception, so the instantiation of the ImplBase object should be enclosed in a `try` statement.

Refer to Chapter15, "Exception Handling" on page 295 for more details.

Writing the Client Application

The following steps have been outlined so far:

1. Defining the IDL interface.
2. Compiling the IDL interface.
3. Implementing the IDL interface.
4. Writing the Server application

The next step involves writing Java clients that access implementation objects through IDL interfaces. The client which follows creates a variable of type `grid` and serves as an illustration. The client calls the static method `bind()` on the IDL-generated `gridHelper` class to create an instance of `grid`. The created object acts as a proxy for an object that implements the `grid` IDL interface in the OrbixWeb server application. A proxy is a local Java object that acts as a representative for a remote object. Its purpose is to transparently forward Java method calls to the remote object.

```
package gridDemo;

import IE.Iona.OrbixWeb._CORBA;
import org.omg.CORBA.SystemException;
import org.omg.CORBA.ORB;

public class javaclient1 {
   public static void main (String args[]) {
      String hostName = null;
1        grid gridProxy = null;

      if (args.length < 1) {
         System.out.println("Usage : javaclient1 [<hostname>]\n");
         System.exit(1);
      }
      else
2        hostName = new String (args[0]);

3        ORB.init (args,null);

      try {
4        gridProxy = gridHelper.bind (":gridServer", hostName);
      }
```

```
            catch (SystemException ex) {
               System.out.println ("Exception caught during bind : "
                                   + ex.toString());
               System.exit (1);
            }
            short width  = 0;
            short height = 0;

            try {
5              width  = gridProxy.width();
               height = gridProxy.height();
            }
            catch (SystemException ex) {
               System.out.println ("FAIL\tException getting the width
                                    and height of the grid.");
               System.out.println (ex.toString());
               System.exit(1);
            }

            System.out.println ("Grid width is  " + width);
            System.out.println ("Grid height is " + height);
            short x_coord = 0;
            short y_coord = 0;

            try {
6              gridProxy.set (x_coord, y_coord, 0);
               x_coord = 2;
               y_coord = 4;
               gridProxy.set (x_coord, y_coord, 123);
            }
            catch (SystemException ex) {
               System.out.println ("FAIL\tException setting values
                                    in grid.");
               System.out.println (ex.toString());
               System.exit (1);
            }

            int val = 0;

            try {
               x_coord = 2; y_coord = 4;
7              val = gridProxy.get (x_coord, y_coord);
            }
```

```
      catch (SystemException ex) {
         System.out.println("FAIL\tException getting value
                             from grid.");
         System.out.println (ex.toString());
         System.exit (1);
      }

      if (val != 123) {
         System.out.println ("FAIL \tThe value " + val + " returned
                               for grid[2,4] is incorrect.");
         System.exit (1);
      }
      System.out.println ("Value for grid[2,4] is " + val);
      System.out.println ("\nGrid demo finished.");
   }
}
```

This code is described as follows:

1. Create a variable to hold a reference to the proxy grid.
2. Extract the host name of the server machine from the command line.
3. Initialize OrbixWeb for an application.
4. Establish a CORBA connection with the grid server and return a proxy for the server grid object. The `bind()` call causes the OrbixWeb daemon to launch the grid server, and enables it to accept remote requests. The first parameter to `bind()` is a string of the following form:

   `<object name>:<server name>`

   This names the object and the server in which the object is running. In this example the object is not named so OrbixWeb can choose any `grid` object in the specified server. The server name, `gridServer`, is the name of the server as registered in the Implementation Repository on the server host. Refer to "Registering the Server" on page 26 for more details.

   The second parameter to `bind()` specifies the host name for the target server.
5. Get the width and height by calling the remote `width()` and `height()` operations on the grid server.
6. Set values in the remote grid.
7. Retrieve a value from the grid.

Compiling the Client and Server

Details of the next step, compiling the client and server, are specific to the Java development environment used. It is possible, however, to describe general requirements. These are illustrated here using the Sun Java Developer's Kit (JDK)[1]. This is the development environment used by the OrbixWeb demonstration makefiles.

To compile an OrbixWeb application, you must ensure that the Java compiler can access the following:

- The Java API classes, located in the `classes.zip` file in the `lib` directory of your JDK installation.
- The `org.omg.CORBA` package, located in the `classes` directory of your OrbixWeb installation.
- The `IE.Iona.OrbixWeb` package, also located in the `classes` directory of your OrbixWeb installation.
- Any other pre-existing classes required by the application.

Compiling the Server Application

To compile the server application, you must invoke the Java compiler on the user generated source files, and on the files generated by the IDL compiler. In the grid server example, the user generated source files are as follows:

- `javaserver1.java`
- `GridImplementation.java`

The IDL generated files are as follows:

- `_gridSkeleton.java`
- `_gridImplBase.java`
- `grid.java`

All of these files are located in the `grid/Java_output/gridDemo` directory. Discussion of the files generated by the IDL compiler is deferred until "OrbixWeb IDL Compilation" on page 29.

1. The JDK version number must be 1.0.2 or higher.

Compiling the Client Application

To compile the client application, invoke the Java compiler on the client source file and on the files generated by the IDL compiler. In the grid client example, the source file is `javaclient1.java` and the generated files are as follows:

- `gridDemo/_gridStub.java`
- `gridDemo/grid.java`

You can use the following example command to compile all Java source files from the command line:

UNIX

```
% /<JDK Location>/bin/javac -classpath/<OrbixWeb Location>
  /classes/<JDK Location>/lib/classes.zip -d /<OrbixWeb
  Location>/classes *.java java_output/gridDemo/*.java
```

Windows

```
> c:\<JDK Location>\bin\javac -classpath c:\<OrbixWeb Location>
  \classes; c:\<JDK Location>\lib\classes.zip -d c:\<OrbixWeb
  Location>\classes *.java java_output\gridDemo\*.java
```

You may use the standard Java command line to compile all the required Java source files, as shown in the preceding command. Alternatively, OrbixWeb provides a convenience tool called `owjavac` that acts as a front end to your chosen Java compiler. This tool passes the default `classpath` and `classes` directory to the compiler, avoiding the need to set environment variables.

The OrbixWeb `demos/grid` directory provides a script which calls `owjavac` as required. To compile the java source files, enter the appropriate command from the `grid` directory:

UNIX `% gmake`

Windows `> compile`

You can use these commands for all of the OrbixWeb demonstrations from the appropriate `demos` directory. These commands run the IDL compiler, compile the Java source files and run the client and server.

For details on the use of the `owjava` and `owjavac` wrapper utilities, see "Using the Wrapper Utilities" on page 240.

Registering the Server

The next step involves registering the server in the Implementation Repository. This allows the server to be launched automatically. The Implementation Repository is a server database that maintains a mapping from the server name to the name of the Java class that implements the server. If the server is registered, it is automatically run through the Java interpreter when a client binds to the `grid` object.

Running the OrbixWeb Daemon

Before registering the server you should ensure that an OrbixWeb daemon process (`orbixd` or `orbixdj`) is running on the server machine.

To run the OrbixWeb Java Daemon type the `orbixdj` command from the `bin` directory of your OrbixWeb installation.

To run the OrbixWeb Daemon, type the `orbixd` command from the `bin` directory of your OrbixWeb installation.

In Windows, you can also start a daemon process by clicking on the appropriate menu item from the OrbixWeb folder.

putit

Once an OrbixWeb daemon process is running, you can register the server. To register the `gridServer`, use the `putit` command as follows:

```
putit -j gridServer gridDemo.javaserver1
```

The `-j` switch indicates that the specified server should be launched via the Java Interpreter. Refer to "The OrbixWeb putit Utility for Server Registration" on page 257 for more details on the `putit` command.

The second parameter to `putit` is the server name, `gridServer` in the grid example. This is also the name of the server passed to `impl_is_ready()` in "Writing the Server Application" on page 18. The third parameter is the name of the class which contains the server's `main()` method, `javaserver1` in the grid example. This is the class which should be run through the Java interpreter.

The server registration step is automated by a script in the `demos\grid` directory which executes the `putit` command.

Running the Client Application

The final programming step involves running the Java interpreter on the bytecode (`.class` files) produced by the Java compiler. When running an OrbixWeb client application, you must ensure that the interpreter can load the following:

- The Java API classes, stored in the `classes.zip` file in the `lib` directory of your JDK installation.
- The `org.omg.CORBA` package, stored in the `classes` directory of your OrbixWeb installation.
- The `IE.Iona.OrbixWeb` package, also stored in the `classes` directory of your OrbixWeb installation
- Any pre-existing classes required by the application.
- Any classes compiled during the client compilation stage.

You can use the `owjava` tool as an alternative to the standard Java command line. This is a wrapper utility that acts as a front-end to your chosen Java interpreter. The `owjava` tool passes the default `classpath` to the interpreter, avoiding the need to set up environment variables. Refer to "Using the Wrapper Utilities" on page 240 for more details on this convenience tool.

To execute the `gridDemo/javaclient1.class`, use the following command:

```
owjava gridDemo.javaclient1 <server host>
```

A script named `javaclient1` in the `demos/grid` directory implements this step. To run the client application use the following command:

```
javaclient1 <server host>
```

Summary of Programming Steps

This section outlined the steps involved in creating a distributed client / server application using OrbixWeb. These steps are as follows:

1. Define the interfaces to objects used by the application, using the CORBA standard IDL.
2. Compile the IDL to generate the Java code.
3. Implement the IDL interface using the generated code.
4. Write a server, using the generated code, to create CORBA objects, and initialize the ORB to receive calls.
5. Write a client application to use the CORBA objects located in the server.
6. Compile the client and server applications.
7. Register the server in the OrbixWeb Implementation Repository.
8. Run the client application.

The next section examines the OrbixWeb IDL compilation process, focusing on the Java classes and interfaces generated by the IDL compiler.

OrbixWeb IDL Compilation

The OrbixWeb IDL compiler produces Java code corresponding to the IDL definition. The mapped Java code consists of code that allows a client to access an object through the `grid` interface, and code that allows a `grid` object to be implemented in a server.

The IDL compilation produces Java constructs (six classes and two interfaces) from the definition of interface `grid`. In compliance with Java requirements, each public class or interface is located in a single source file with a `.java` suffix. Each source file is located in a directory that follows the Java mapping for package names to directory structures.

By default, the OrbixWeb IDL compiler creates a local `java_output` directory into which the generated Java directory structure is placed. An alternative target directory can be specified using the compiler switch `-jO`.[2]

Each generated file contains a Java class or interface that serves a specific role in an application.

Client-Side Mapping	**Description**
`grid`	A Java interface whose methods define the Java client view of the IDL interface.
`_gridStub`	A Java class that implements the methods defined in interface `grid`. This class provides functionality which allows client method calls to be forwarded to a server.
Server-Side Mapping	**Description**
`_gridSkeleton`	A Java class used internally by OrbixWeb to forward incoming server requests to implementation objects. You do not need to know the details of this class.
`_gridImplBase`	An abstract Java class that allows server-side developers to implement the `grid` interface using the ImplBase approach.

2. Refer to Appendix A, "IDL Compiler Switches" on page 505, for details of IDL Compiler switches.

`_tie_Grid`	A Java class that allows server-side developers to implement the `grid` interface using delegation. This approach to interface implementation is called the TIE approach. The TIE approach is an OrbixWeb-specific feature, and is not defined in the CORBA specification.
`_gridOperations`	A Java interface, used in the TIE approach only, that maps the attributes and operations of the IDL definition to Java methods. These methods must be implemented by a class in the server, following the TIE approach.

Client and Server Side Mapping	**Description**
`gridHelper`	A Java class that allows you to manipulate IDL user-defined types in various ways.
`gridHolder`	A Java class defining a `Holder` type for class `grid`. This is required for passing `grid` objects as `inout` or `out` parameters to and from IDL operations. See "Holder Classes" on page 95.
`gridPackage`	A Java package used to contain any IDL types nested within the `grid` interface, for example, structures or unions.

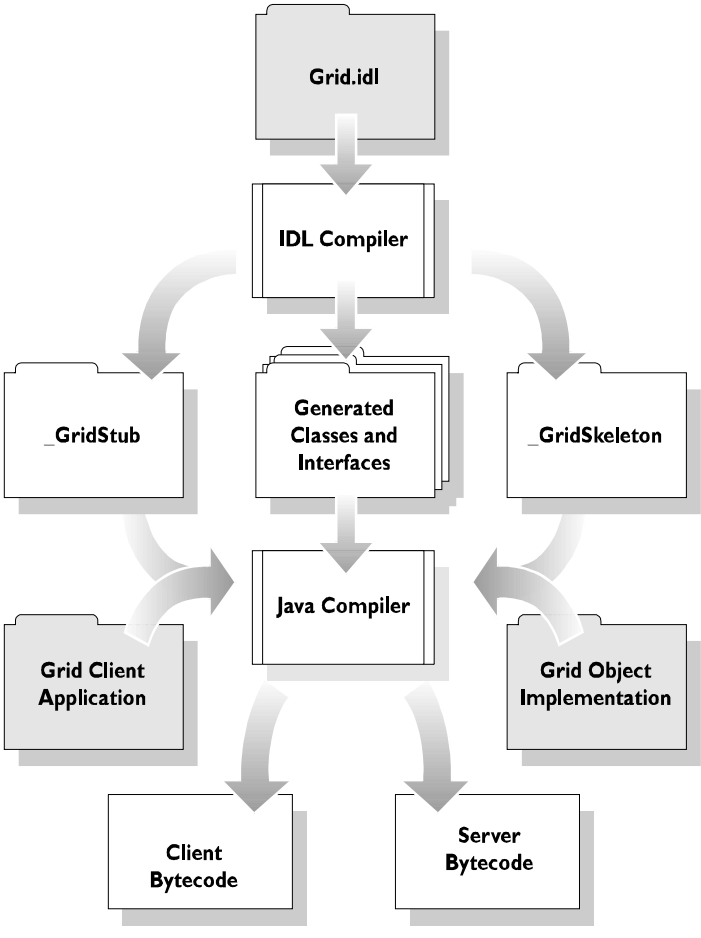

Figure 6: *Overview of the Compilation of the Grid Interface*

Examining the Roles of the Generated Interfaces and Classes

The relationships between the Java types produced by the IDL compiler can be illustrated by a brief examination of the generated source code. The IDL to Java compiler maps the IDL interface `grid` to a Java interface of the same name. A corresponding Java class (`_gridStub`) implements this Java interface.

Client-Side Mapping

The Java files `grid.java` and `_gridStub.java` support the client-side mapping. The `grid.java` file maps the operations and attributes in `gridDemo.idl` to Java methods as follows:

```
// Java generated by the OrbixWeb IDL compiler
//
public interface grid
   extends org.omg.CORBA.Object {
      public short height();
      public short width();
      public void set(short row, short col, int value);
      public int get(short row, short col);
}
```

This Java interface defines an OrbixWeb client view of the IDL interface defined in `gridDemo.idl`. The Java interface is implemented by the Java class `_gridStub` in the file `_gridStub.java` as follows:

```
// Java generated by the OrbixWeb IDL compiler

public class _gridStub
   extends org.omg.CORBA.portable.ObjectImpl
   implements grid {
      public short height() {
         ...
      }
      public short width() {
         ...
      }
      public void set(short row,short col,int value) {
         ...
      }
```

```
        public int get(short row,short col) {
           ...
        }
        ...
        }
}
```

The primary role of the `_gridStub` Java class is to transparently forward client invocations on `grid` operations to the appropriate implementation object in the server. The IDL is mapped to the Java interface `grid` to allow for multiple inheritance. The implementation is then supplied by the corresponding `_gridStub`.

The `set()` and `get()` IDL operations are mapped to corresponding Java methods. The parameters, which are IDL basic types in the IDL definition, are mapped to equivalent Java basic types. For example, the IDL type `long` (a 32-bit integer type) maps to the Java type `int` (also a 32-bit integer type). For IDL types that have no exact Java equivalent, an approximating class or basic type is used. Refer to "IDL to Java Mapping" on page 91.

The two `readonly` attributes are mapped with the Java methods (`height()` and `width()`). These attributes are not mapped to Java public member variables because the server and client may not be stored in the same address space.

Server-Side Mapping

OrbixWeb provides support for two approaches to implementing an IDL interface:

- The *ImplBase* approach, which uses inheritance:

  The generated Java class used in the ImplBase approach is `_gridImplBase`.

  The ImplBase approach is used in this chapter to implement the `grid` IDL interface.

- The *TIE* approach, which uses delegation.

  The generated Java constructs used in the TIE approach are the interface `_gridOperations` and the class `_tie_Grid`.

The grid example illustrated in this chapter uses the ImplBase approach, the standard CORBA approach. The use of the TIE and ImplBase approaches is discussed in detail in "Implementing the Interfaces" on page 138. The TIE approach, which uses delegation, is preferred for many Java applications and applets. This approach is used to implement the `account` example in Chapter 7, "Using and Implementing IDL Interfaces" on page 135.

After the IDL interface has been implemented, a server creates an instance of the implementation class. This server then connects the created object to the ORB runtime which passes incoming invocations to the implementation object.

3

Getting Started with Java Applets

This chapter extends the grid example from Chapter 2 to take account of a common form of distributed Java system: a downloadable client applet which communicates with a back-end server. You should be familiar with the material covered in Chapter 2, "Getting Started with Java Applications" before continuing with this chapter.

Review of OrbixWeb Programming Steps

Recall the programming steps typically required to create a distributed client/server application with OrbixWeb:

1. Define the interfaces to objects used by the application, using the CORBA standard Interface Definition Language (IDL).
2. Generate Java code from the IDL using the IDL compiler.
3. Implement the IDL interface, using the generated code.
4. Write a server that creates instances of the generated classes and informs OrbixWeb when initialization is complete.
5. Write a client application that connects to the server and uses server objects.
6. Compile the client and server applications.
7. Register the server in the Implementation Repository.
8. Run the client application.

This chapter uses the example IDL interface for a two-dimensional grid outlined in "Defining the IDL Interface" on page 14. The sample code described in this chapter is available in the `demos/gridApplet` directory of your OrbixWeb installation.

Providing a Server

This chapter illustrates a distributed architecture in which a downloadable client applet communicates with an OrbixWeb server through an IDL interface. This client-server architecture is a common requirement in the Java environment where small, dynamic client applets may be downloaded to communicate with large, powerful back-end service applications. Architectures in which full OrbixWeb servers are coded as downloadable applets are less common, and are not described here.

The example server used in this chapter is developed in "Writing the Server Application" on page 18. OrbixWeb programming steps 1 to 4 are essentially identical for Java applications and for Java applets (see "Review of OrbixWeb Programming Steps" on page 35). The main differences between programming for Java applications and for Java applets occurs with step 5, writing the client.

Writing a Client Applet

This programming step is equivalent to step 5, "Writing the Client Application" on page 21. In this section, a simple Java applet is developed, providing a graphical user interface to the IDL interface `grid`. This example builds upon the concepts introduced in the grid client example.

Writing the client applet can be broken down into the following four sub-steps, each of which corresponds to a particular demonstration source file:

1. Creating the user interface (`GridPanel.java`).
2. Adding OrbixWeb client functionality (`GridEvents.java`).
3. Creating the applet (`GridApplet.java`).
4. Adding the client to a HTML file (`GridApplet.html`).

These files are located in the `demos/gridApplet` directory of your OrbixWeb installation. The package name for the Java classes in this example is `gridAppletDemo`. It is assumed that the IDL file `gridAppletDemo.idl` was compiled with the following command:

```
idl -jP gridAppletDemo gridAppletDemo.idl
```

The development of an OrbixWeb client can be completely decoupled from the server-side development process. For this reason, when compiling the IDL file, the package name chosen for the client may be different from the package name for the server.

Creating the User Interface

The grid applet graphical user interface (Figure 7 on page 39) consists of two sections:

Server Details	This section allows users to specify a target OrbixWeb server.
Object Details	This section allows a `grid` object to be queried and updated.

The GUI source code in `GridPanel.java` uses the Java Abstract Windowing Toolkit package (`java.awt`) to create and arrange each of the elements within a `java.awt.Panel` container. You should refer to your Java documentation for details of the AWT. The code sample which follows gives the names of individual GUI components, such as buttons and text fields. The details of how the GUI is implemented are not discussed here:

```
// Java
// In file gridAppletDemo/GridPanel.java.
package gridAppletDemo;
import java.awt.*;

public class GridPanel extends Panel {
   // Button string constants.
   final String conBStr = "Connect";
   final String disBStr = "Disconnect";
   final String dimBStr = "Get Grid Dimensions";
   final String getBStr = "Get Cell Value";
   final String setBStr = "Set Cell Value";

   // Components for Server Details section
   // bind() labels.
   Label nameL;
   Label hostL;

   // bind() text fields.
```

```
    TextField nameField;
    TextField hostField;
    // bind() buttons.
    Button connectButton;
    Button disconnectButton;

    // Components for Object Details section
    // operation labels.
    Label dL;
    Label xL;
    Label yL;
    Label vL;

    // Operation text fields.
    TextField dField;
    TextField xField;
    TextField yField;
    TextField vField;

    // Operation buttons.
    Button getDButton;
    Button getVButton;
    Button setVButton;

    // Sub panels.
    Panel bindPanel;
    Panel botPanel;

    // Constructor.
    public GridPanel () {
    ...
    }
}
```

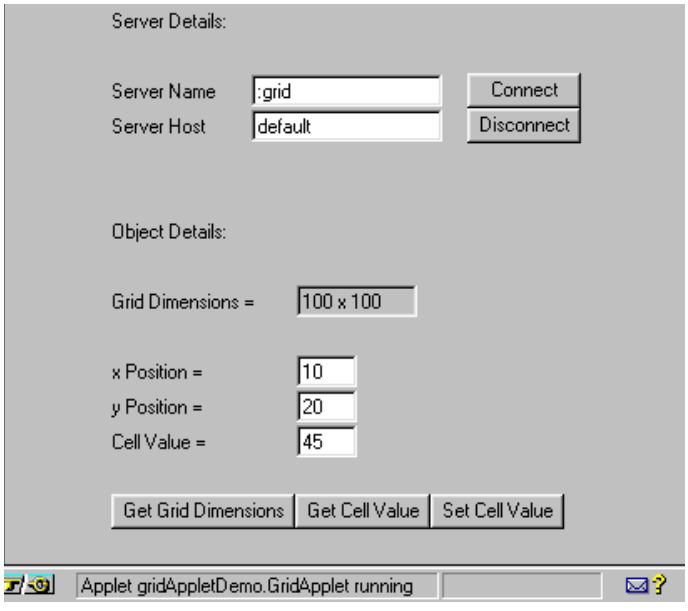

Figure 7: *The Grid Applet Graphical User Interface*

Adding OrbixWeb Client Functionality

In the grid applet example, all OrbixWeb client functions are initiated by GUI button clicks. For the purposes of illustration, the applet maps GUI button clicks directly to individual operations on a `grid` object. Operation parameter values and results are sent and returned using text boxes. This allows the client to receive notification of a button click event, and to determine which button received the event. The client can then react by calling the appropriate operation on a grid proxy object.

A subclass of `GridPanel` named `GridEvents` serves as the container for the various buttons and text fields. The following is an outline of the source code for the class `GridEvents`. The button implementation methods outlined here are expanded on later in this section:

```
// Java
// In file gridAppletDemo/GridEvents.java.

package gridAppletDemo;

import java.awt.*;
import java.lang.*;
import org.omg.CORBA.SystemException;
import org.omg.CORBA.ORB;

public class GridEvents extends GridPanel {
   // grid proxy object
   public grid gRef;

   public GridEvents() {
      super();
      ORB.init(this,null);
      nameField.setText(":gridServer");
   }

   // Notify appropriate method for action event.
   public boolean action (Event event, Object arg) {
      if (conBStr.equals (arg)) {
         bindObject ();
      }
      else if (disBStr.equals (arg)) {
         gRef = null;
         displayMsg ("Disconnected.");
```

```
            }
            else if (dimBStr.equals (arg)) {
               getHeightAndWidth ();
            }
            else if (getBStr.equals (arg)) {
               getCellValue ();
            }
            else if (setBStr.equals (arg)) {
               setCellValue ();
            }
            return true;
         }

      // Connect button implementation.
      public void bindObject() {
         // Details are described
         // later in this section.
      }

      // Get Grid Dimensions button implementation.
      public void getHeightAndWidth() {
         // Details are described
         // later in this section.
      }
      // Get Cell Value button implementation.
         public void getCellValue() {
            // Details are described
            // later in this section.
         }

         // Set Cell Value button implementation.
         public void setCellValue() {
            // Details will be described
            // later in this section.
         }
         ...
      }
```

The grid applet example provides methods to handle the client functionality required for the GUI buttons shown in Figure 7 on page 39. The following section explains each button implementation in detail.

Server Details

The **Server Details** section of the grid applet GUI includes the following buttons:

- **Connect**
- **Disconnect**

Connect

The **Connect** button functionality is implemented by the method `bindObject()`:

```
public void bindObject() {
   String tmp;
   String markerServer;
   String hostName;

   // get server name from text field
   if ((tmp = nameField.getText()) == null) {
      markerServer = "";
   }
   else
      markerServer = ":" + tmp;

   // get host name from text field
   hostName = hostField.getText();

   // bind to server object
   try {
      gRef = gridHelper.bind
         (markerServer,hostName);
   }
```

```
        catch(SystemException se) {
           displayMsg("Connect failed.\n" + "Unexpected
                       exception:\n" + se.toString());
           return;
        }
        displayMsg("Connect succeeded.");
     }
```

The **Connect** button forces the client to bind to a `grid` object in the server specified by the **Server Name** and **Server Host** text fields. The `bind()` method creates a proxy object of type `grid` and binds it to an implementation object in the specified server. No object marker is specified in the `bind()` call, so OrbixWeb is free to choose *any* `grid` object in that server.

Disconnect

The **Disconnect** button is implemented by the following line of code in the `action()` method:

```
gRef = null;
```

This button allows the user to destroy a previously created proxy object by assigning it to the Java value `null`. This does not actually close the connection; to do this, you must call the following:

```
_CORBA.Orbix.closeConnection(gRef);
```

Object Details

The **Object Details** section of the grid applet GUI includes the following buttons:

- **Get Grid Dimensions**
- **Get Cell Value**
- **Set Cell Value**

These buttons allow `grid` operations to be called on a proxy object created by the **Connect** button. The methods which implement these buttons call the proxy member variable `gRef`.

Get Grid Dimensions

The **Get Grid Dimensions** button is implemented as follows:

```
public void getHeightAndWidth() {
   short h, w;

   // check that proxy exists
   if (gRef == null) {
      displayMsg("Get dimensions failed - not
                           connected to server.");
      return;
   }

   // call attribute methods
   try {
      h = gRef.height();
      w = gRef.width();
   }
   catch (SystemException se) {
      displayMsg("Get dimensions failed.\n" +"Unexpected
                        exception:\n" + se.toString());
      return;
   }
   dField.setText(Integer.toString(w) + " x "
                           + Integer.toString(h));
   displayMsg("Get dimensions succeeded.");
}
```

Get Cell Value

The **Get Cell Value** button is implemented as follows:

```
public void getCellValue() {
   short x, y;
   int cellVal = 0;

   // check that proxy exists
   if (gRef == null) {
      displayMsg("Get cell value failed - not
                           connected to server.");
      return;
   }
```

```
      // get position from text fields
      try {
         x = (short) Integer.parseInt (xField.getText());
         y = (short) Integer.parseInt (yField.getText());
      }
      catch (java.lang.NumberFormatException nfe) {
         displayMsg("Get cell value failed - " +
                       "invalid co-ordinate values.\n");
         return;
      }

      // call get operation
      try {
         cellVal = gRef.get (x, y);
      }
      catch (SystemException se) {
         displayMsg("Get cell value failed.\n"+"Unexpected
                           exception:\n" + se.toString());
         return;
      }
      vField.setText(Integer.toString (cellVal));
      displayMsg("Get cell value succeeded.");
   }
```

Set Cell Value

The **Set Cell Value** button has the following implementation:

```
   public void setCellValue() {
      short x, y;
      int cellVal;

   // check that proxy exists
      if (gRef == null) {
         displayMsg("Set cell value failed - not
                              connected to server.");
      return;
      }
```

```
        // get position and value from text fields
        try {
           x = (short) Integer.parseInt (xField.getText());
           y = (short) Integer.parseInt (yField.getText());
           cellVal = Integer.parseInt (vField.getText());
        }
        catch (java.lang.NumberFormatException nfe) {
           displayMsg("Set cell value failed - " + "invalid
                                co-ordinate or cell value.");
           return;
        }

     // call set operation
        try {
           gRef.set(x, y, cellVal);
        }
        catch (SystemException se) {
           displayMsg("Set cell value failed.\n" +"Unexpected
                              exception:\n" + se.toString());
           return;
        }
        displayMsg("Set cell value succeeded.");
     }
```

Error Handling: Integration with Java Exceptions

In the example described in "Writing the Client Application" on page 21, OrbixWeb system exceptions are handled in `catch` clauses by displaying the exception `toString()` output in the `System.out` print stream. This information is helpful when you are debugging OrbixWeb clients. In a client applet, however, it may not be practical to output the information to a print stream. In this example, exception strings are displayed in information dialog boxes. The file `MsgDialog.java` implements a generic dialog class for this purpose:

```
// In file gridAppletDemo/MsgDialog.java.
package gridAppletDemo;

import java.awt.*;

public class MsgDialog extends Frame {
   protected Button button;
   protected Msg label;
```

```
        public MsgDialog(String title,String message) {
           // Details omitted.
        }
           // Other class details omitted.
    }
```

The details of this class implementation is not important. OrbixWeb error-handling can be added to the `GridEvents` class by defining a display method as follows:

```
void displayMsg (String msg) {
   MsgDialog msgDlog
      = new gridAppletDemo.MsgDialog
         ("Grid Operation Result", msg);
   msgDlog.resize (380, 200);
   msgDlog.show ();
}
```

This allows any string, including system exception strings, to be displayed in a dialog box. Figure 8 shows a dialog box displays a communications failure exception for a host called "default" on port number 2002.

Figure 8: *System Exception Dialog Box*

Creating the Applet

To create the grid client applet, define a subclass of `java.applet.Applet` and add a `GridEvents` object to this class:

```
// Java
// In file gridAppletDemo/GridApplet.java.

package gridAppletDemo;

import org.omg.CORBA.SystemException;
import org.omg.CORBA.INITIALIZE;
import java.applet.*;
import java.awt.*;
import org.omg.CORBA.ORB;

public class GridApplet extends Applet {
   // main display panel
   GridEvents gridEvents;

public void init() {
     // initialize the ORB
     // This call is essential for applets.
   try {
     ORB.init (this, null);
   }
   catch (INITIALIZE ex) {
     System.err.println("failed to initialize:"+ex);
   }
   gridEvents = new GridEvents();
   // Add panel to applet
   this.add (gridEvents);
}
```

ORB initialization

Because OrbixWeb uses the standard OMG IDL to Java mapping, all client and server applets must call `org.omg.CORBA.ORB.init()` to initialize the ORB. This returns a reference to the `ORB` object. You can then invoke the ORB methods defined by the standard on this instance.

The example applet, `GridApplet.java`, uses the following version of `org.omg.CORBA.ORB.init()`:

```
ORB.init(Applet app, java.util.Properties props)
```

You must use this version of `init()` for applet initialization. In the example, the client applet passes a reference to itself using the `this` parameter. The `props` parameter, used to set configuration properties, is set to `null`. This means that the default system properties are used instead.

This version of the `init()` method returns a new fully functional ORB Java object each time it is called. Refer to the *OrbixWeb Programmer's Reference* for further information on class `org.omg.CORBA.ORB` and `ORB.init()`.

Adding the Applet to a HTML File

In HTML terms, an OrbixWeb applet client behaves exactly like a standard Java applet. It can be included in a HTML file using the standard <APPLET> tag. The source for `GridApplet.html` serves as an example:

```
// HTML
// in file GridApplet.html

<HTML>
<HEAD>
   <TITLE>OrbixWeb grid applet demo</TITLE>
</HEAD>

<BODY>
   <H1>Grid Client</H1>

   <APPLET code="gridAppletDemo/GridApplet.class"
1     codebase="../../classes/"
      width=390 height=560>
   <param name="org.omg.CORBA.ORBClass"
2     value="IE.Iona.OrbixWeb.CORBA.ORB>
   <param name="org.omg.CORBA.ORBSingletonClass"
      value="IE.Iona.OrbixWeb.CORBA.ORB>
   </APPLET>
</BODY>
</HTML>
```

1. The codebase attribute of the HTML <APPLET> tag indicates the location of the additional classes required by the applet.
2. Pass the parameter value `IE.Iona.OrbixWeb.CORBA.ORB` to enable use of the OrbixWeb ORB implementation. This means that OrbixWeb specific methods such as `bind()` can be used.

Note: If you wish to use callbacks, you should pass the package `IE.Iona.OrbixWeb.CORBA.BOA`, instead of `IE.Iona.OrbixWeb.CORBA.ORB`. Refer to Chapter 17, "Callbacks from Servers to Clients" for more details on using callbacks with OrbixWeb.

Compiling the Client Applet

The instructions for compiling an OrbixWeb applet are identical to those for a standard OrbixWeb application, as described in "Compiling the Client and Server" on page 24.

You must ensure that the Java compiler can access the Java API packages (including `java.awt` for this sample code), the OrbixWeb `IE.Iona.OrbixWEB.CORBA` package, and any applet-specific classes. Invoke the compiler on all the Java source files for the application.

The following files are required for the grid applet example:

- `_GridStub.java`
- `Grid.java`
- `GridPanel.java`
- `GridEvents.java`
- `GridApplet.java`
- `MsgDialog.java`
- `Msg.java`

The OrbixWeb `demos/gridApplet` directory provides a script which invokes the `owjavac` wrapper utility as required. To compile the client applet, type the appropriate command at the operating system prompt:

UNIX `% gmake`

Windows `> compile`

Running the Client Applet

When running the client applet, you must use a Web browser or an applet viewer to view the HTML file. For example, you can use the JDK `appletviewer` as follows:

```
appletviewer GridApplet.html
```

Java applets differ slightly from standalone Java applications in their requirements for access to class directories. Before running the viewer, you can specify the locations of required classes in the CLASSPATH environment variable. The classes required are identical to those for an OrbixWeb client application:

- The Java API classes, stored in the `classes.zip` file in the `lib` directory of your JDK installation.
- The `org.omg.CORBA` package, stored in the `classes` directory of your OrbixWeb installation.
- The `IE.Iona.OrbixWeb` package, also stored in the `classes` directory of your OrbixWeb installation.
- Any pre-existing classes required by the application.
- Any classes compiled during the client compilation stage.

An alternative approach is to provide access to all the classes the applet requires in a single directory. Instead of setting environment variables, you can use the CODEBASE attribute of the HTML <APPLET> tag to indicate the location of the required classes. This is approach is recommended, and is the approach used in "Creating the Applet" on page 48.

The configuration file, `OrbixWeb.properties` is loaded from the location specified by the CODEBASE attribute of the <APPLET> tag. If you do not specify the CODEBASE attribute, the directory containing HTML file is used as the default location.

Refer to Chapter 4, "Getting Started with OrbixWeb Configuration" on page 53 for more details on `OrbixWeb.properties`.

Security Issues for Java Applets

Java applets are subject to important security restrictions, imposed by the Java environment and by Web browsers. The severity of these restrictions is often dependent on browser technology. See Chapter 13, "Using OrbixWeb on the Internet" on page 249 for further information.

4

Getting Started with OrbixWeb Configuration

The OrbixWeb Configuration Tool (`owconfig`*) allows you to change the default configuration settings for OrbixWeb using a graphical user interface. The Configuration Tool edits* `OrbixWeb.properties` *and* `Orbix.cfg`*. These files store the configuration settings of your OrbixWeb installation.*

You may need to change default configuration settings for a variety of reasons, including the following:

- Enabling or disabling particular functionality.
- Modifying the location of the Implementation Repository.
- Changing the JDK version used by the OrbixWeb daemon.
- Changing specific port numbers used.

You can use the OrbixWeb Configuration Tool to make these configuration changes. Refer to the chapter, "OrbixWeb Configuration" in the *OrbixWeb Programmer's Reference* for a full description of OrbixWeb configuration parameters.

OrbixWeb Configuration Files

OrbixWeb uses the following configuration files:

- `OrbixWeb.properties`
- `Orbix.cfg`

OrbixWeb.properties

Every OrbixWeb client and server applet or application must have access to `OrbixWeb.properties`. This file holds the configuration information for the Java components of OrbixWeb. The Java daemon (`orbixdj`) will not start without this file. The installation process creates a default `OrbixWeb.properties` file, located in the `classes` directory of your OrbixWeb installation.

The OrbixWeb Configuration Tool reads its settings from `OrbixWeb.properties`.

Orbix.cfg

`Orbix.cfg` holds configuration information for the natively compiled executable components of OrbixWeb, such as the IDL compiler and `orbixd`. The installation process creates a default `Orbix.cfg` file, located in your OrbixWeb installation directory.

The Configuration Tool does *not* read settings from `Orbix.cfg`, it uses `OrbixWeb.properties` as the master set of configuration settings. The Configuration Tool does however modify both `OrbixWeb.properties` and `Orbix.cfg` to reflect configuration changes made.

Note: The Configuration Tool is the recommended approach to OrbixWeb configuration. Direct editing of `Orbix.cfg` or `OrbixWeb.properties` is strongly discouraged, because common text editors corrupt these files.

Configuration Tool Requirements

To use the Configuration Tool, you must have correct settings for three basic configuration items:

- the Java interpreter
- the Java compiler
- the CLASSPATH

See "The General Page" on page 56 for details of how to set these values.

If your `OrbixWeb.properties` file becomes corrupt or is deleted, these fundamental settings will no longer be available to the Configuration Tool. You will then need to hand-edit a minimal `OrbixWeb.properties` file. This file should be placed in the `classes` directory of your OrbixWeb installation. The required file format is as follows:

```
OrbixWeb.IT_JAVA_INTERPRETER=<path to Java interpreter>
OrbixWeb.IT_JAVA_COMPILER=<path to Java compiler>
OrbixWeb.IT_DEFAULT_CLASSPATH=<class path including JDK and
                                        OrbixWeb classes>
```

Starting the OrbixWeb Configuration Tool

To start the OrbixWeb Configuration Tool from the command line, type the following command:

UNIX	`owconfig`
Windows	`OWConfig`

Alternatively, select the OrbixWeb Configuration Tool from the Windows Program Menu.

When you start the Configuration Tool it automatically loads its settings from the default `OrbixWeb.properties` file in the `classes` subdirectory. If `OrbixWeb.properties` can not be found, a dialog box is displayed to enable you to find this file.

The Configuration Tool Main Panel

The Configuration Tool main panel, as shown in Figure 9 on page 57, consists of three main sections:

Menu Bar	The menu bar at the top of the screen provides access to the **File** menu. This enables you to load and save configurations. To save changes to `OrbixWeb.properties`, select **File | Save** or **File | Save as** from the Configuration Tool main menu button. To save changes to `Orbix.cfg`, select **File | Save Orbix.cfg as** from the Configuration Tool main menu button.
Tabbed Folder	The pages of the tabbed folder in the central section of the screen hold all the OrbixWeb configuration values which can be changed.
Status Bar	The status bar at the bottom of the screen provides user feedback on the actions performed by the tool.

The General Page

If you wish to change the Java tool kit you are using, you need to change the first three settings on the General page of the Configuration Tool, as shown by Figure 9 on page 57. To change these settings, enter the path in the appropriate text box:

- **Java Interpreter**
- **Java compiler**
- **Default class path**

You can set how your OrbixWeb installation identifies itself on the network by entering the following values:

- **Hostname**
- **DNS domain name**

Typically, you do not need to set the **Hostname** parameter, this is automatically determined at runtime. However, this can be useful if you wish to control which interface incoming connections will be accepted on. The **DNS domain name** parameter allows you to specify your domain name. This should be set if you plan to use OrbixWeb outside your own domain.

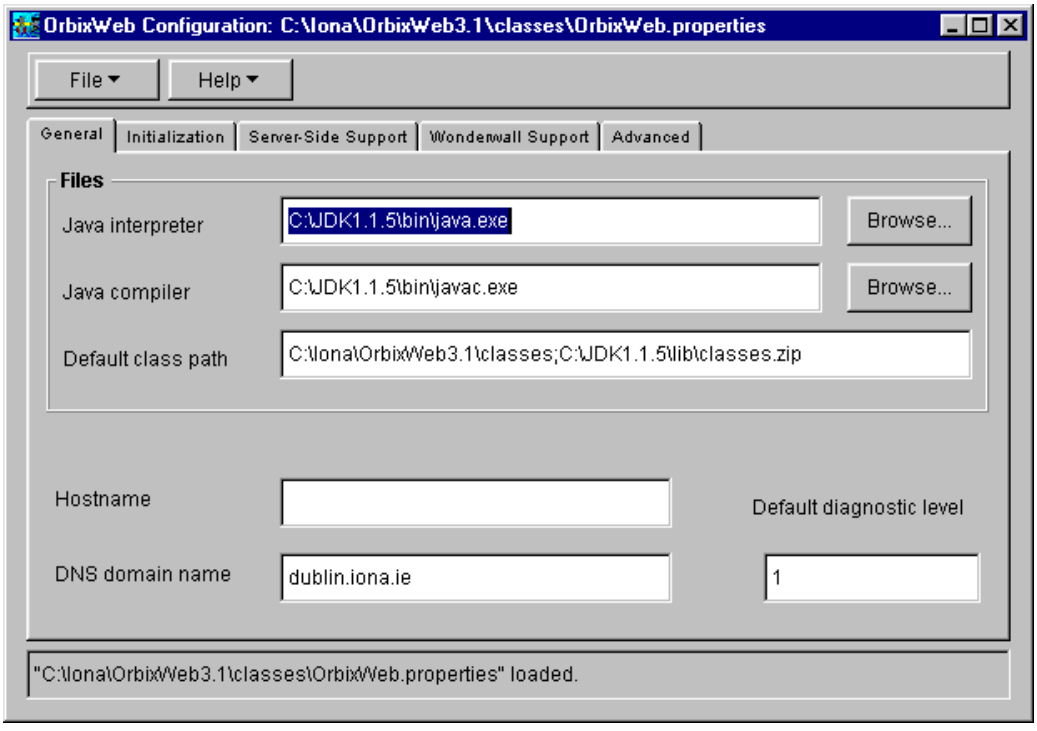

Figure 9: *OrbixWeb Configuration Tool Main Panel*

The **Default diagnostic level** value controls what diagnostic information OrbixWeb logs when calls are made or received. To set the diagnostic level, enter a value in the range `0-255`.

The Initialization Page

The **bind() Support and Activator Setup** section of the initialization page, as shown in Figure 10 on page 59, controls the following configuration settings:

- The ports used by OrbixWeb during `bind()`.
- The port range and Implementation Repository used by the OrbixWeb daemon.
- The behaviour of the bind mechanism itself.

The OrbixWeb daemon waits for incoming connections on the following well-known internet ports:

OrbixWeb daemon Port	The daemon uses this port to communicate with other processes using either the Orbix protocol or IIOP (Internet Inter-Operability Protocol).
OrbixWeb daemon IIOP Port	This port is provided to support legacy daemons which require a separate port for each protocol when listening for incoming connections. To enter a value in this text box, first check **Support legacy daemons with two ports**.

Server processes launched by the daemon are assigned a port number. The daemon uses the defined **Server port range** when allocating ports.

Typically, you do not need to change the settings for this section unless there is a conflict with another installation of Orbix or OrbixWeb on the same machine.

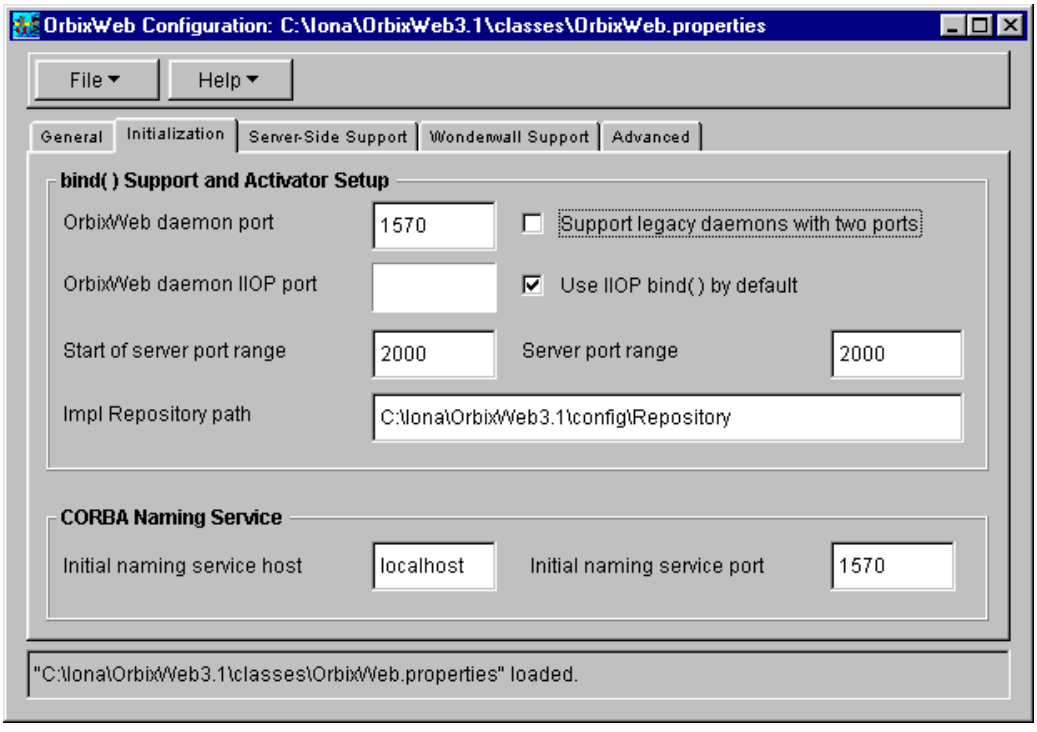

Figure 10: *Configuration Tool Initialization Page*

The **CORBA Naming Service** section allows you to set the host and port that OrbixWeb uses when it tries to contact a Naming Service.

The Server-Side Support Page

There are two sections on the **Server-Side Support** page:

ORB This section allows you configure for client-side support only. For example, for an applet with no interfaces defined that only acts as a client.

BOA This section allows you configure for full client-side and server-side support.

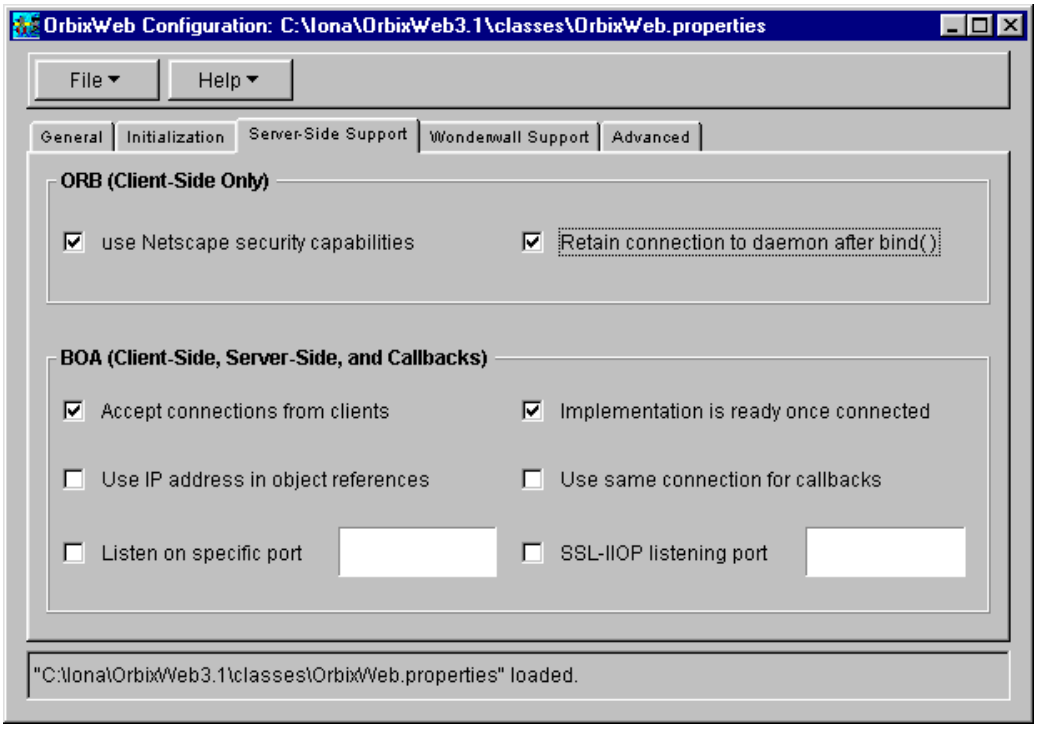

Figure 11: *Configuration Tool Server-Side Support Page*

The **BOA** section enables you to specify the following:

- The port on which the server listens for calls.
- Whether the server accepts connections from clients.
- Whether a client using callbacks supports the connection it established to receive callbacks.
- Whether IP addresses should be used in your object references instead of hostnames. This can be useful if your clients do not have DNS name resolution capability.

The Wonderwall Support Page

Using this page, you can the specify the location of your Wonderwall. This is the IIOP firewall and intranet request routing proxy server provided by OrbixWeb. When an applet using this configuration is downloaded by a client, it can connect to your server via the Wonderwall.

OrbixWeb provides support for two connection mechanisms:

IIOP Proxy	IIOP Proxy support means that OrbixWeb sends an object reference to the Wonderwall. This enables communication with the target object through the Wonderwall using IIOP.
HTTP Tunnelling	HTTP Tunnelling involves wrapping the IIOP Proxy inside HTTP. This means that the object reference can pass through any applet-side firewall which has a HTTP proxy. However, HTTP Tunnelling is not as efficient as pure IIOP, and does not support callbacks from the server to the applet.

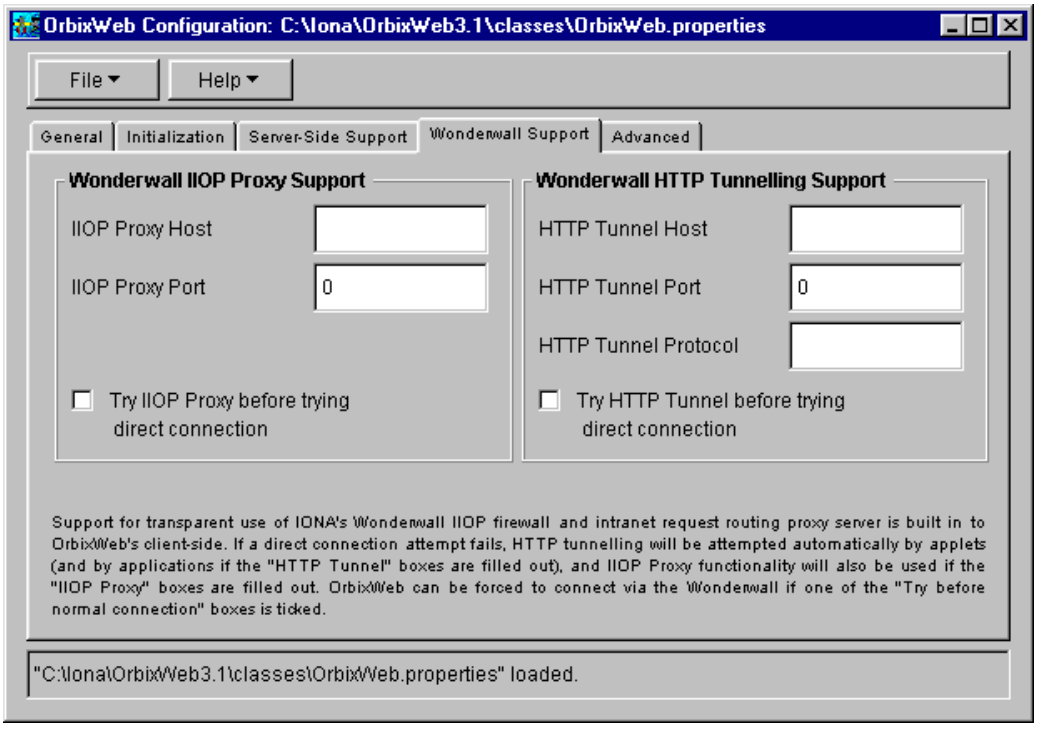

Figure 12: *Configuration Tool Wonderwall Support Page*

To receive callbacks on your OrbixWeb applets from the server, you must set the **Use same connection for callbacks** check box on the Server-Side Support page.

Usually, an applet will try to connect via the Wonderwall only if a direct connection fails. You may wish to specify that an applet using a particular configuration should try to connect via the Wonderwall before trying to connect directly. You can specify this by setting **Try IIOP Proxy before trying connection** and **Try Tunnel before trying direct connection** check boxes.

The Advanced Page

The Advanced page allows you to set the following:

- The size of buffers and tables used by OrbixWeb.
- **Connection keepalive**.
- Miscellaneous settings.

The **Connection keepalive** setting controls how long a connection needs to idle before being closed down automatically by the client-side ORB. If you set this to a value of `-1` milliseconds, connections will never be closed.

Refer to the chapter "OrbixWeb Configuration" in the *OrbixWeb Programmer's Reference* for a full description of OrbixWeb configuration parameters.

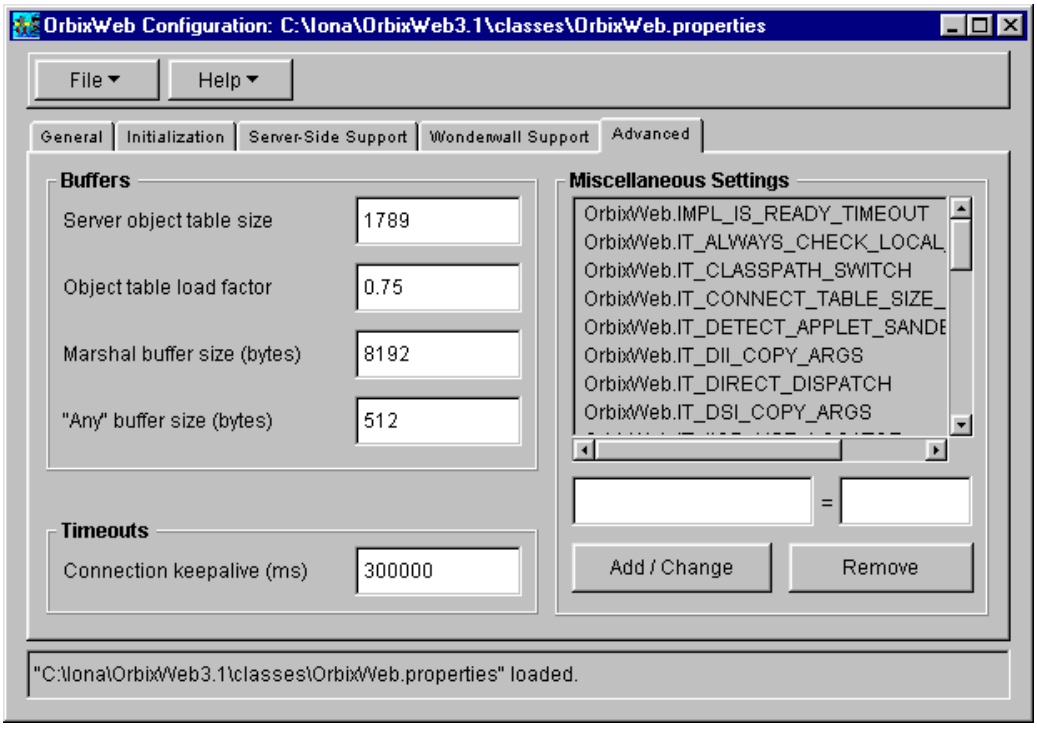

Figure 13: *Configuration Tool Advanced Page*

Learning more about OrbixWeb

Parts II to VI of this guide describe OrbixWeb features in more detail, expand on the information presented in Part I, and:

- Present an overview of the structure of distributed applications.
- Introduce IDL and the corresponding mapping of IDL to the Java programming language. Both client and server programmers must be familiar with this mapping.
- Present further examples of using OrbixWeb to define an interface to a system component and write client and server programs.
- Explain the use of inheritance when defining IDL interfaces, allowing an interface to be defined by extending others.
- Give more details on compiling IDL definitions, registering servers, and configuring OrbixWeb to suit a particular environment.
- Explain that operation calls can be made in more ways than those shown in the overview presented here. Some applications, such as browsers, need to be able to use all of the interfaces defined in a system—even those interfaces which did not exist when the browser was compiled. OrbixWeb supports such applications via its Dynamic Invocation Interface.

Parts V and VI of this guide discuss advanced features which extend the power of OrbixWeb:

- *Filters* can be installed in the system to allow programs to monitor or control incoming or outgoing requests.
- A *proxy* is a local representative or stand-in for a remote object. A smart proxy is an intelligent stand-in. Smart proxies can be written to optimize the performance of a component as perceived by a client.
- To facilitate applications such as browsers, the interface of an object can be examined at runtime (using the *Interface Repository*).

- If OrbixWeb fails to find an object being sought by a client or server, it will informs *loader* objects, which are given the opportunity to load the object from some persistent store. Interfacing OrbixWeb to a persistent store therefore involves writing a loader object and installing this within programs that directly use that persistent store. As a result, OrbixWeb is not tied to using any specific persistent store from a particular vendor.
- OrbixWeb has an inbuilt mechanism for searching the distributed system for a server. If this mechanism is not appropriate or if it needs to be augmented, you can write a *locator* object and install this in preference to the default one.

A full description of the API to OrbixWeb is supplied in the *OrbixWeb Programmer's Reference*.

Part II

CORBA Programming with OrbixWeb

5

Introduction to CORBA IDL

The CORBA Interface Definition Language (IDL) is used to define interfaces to objects in your network. This chapter introduces the features of CORBA IDL and illustrates the syntax used to describe interfaces.

The first step in developing a CORBA application is to define the interfaces to the objects required in your distributed system. To define these interfaces, you use CORBA IDL.

IDL allows you to define interfaces to objects without specifying the implementation of those interfaces. To implement an IDL interface you must do the following:

1. Define a Java class which can be accessed through the IDL interface.
2. Create objects of that class within an OrbixWeb server application.

You can implement IDL interfaces using any programming language for which an IDL mapping is available. An IDL mapping specifies how an interface defined in IDL corresponds to an implementation defined in a programming language. CORBA applications written in different programming languages are fully interoperable.

CORBA defines standard mappings from IDL to several programming languages, including C++, Java, and Smalltalk. The OrbixWeb IDL compiler converts IDL definitions to corresponding Java definitions, in accordance with the standard IDL to Java mapping.

IDL Modules and Scoping

An IDL module defines a naming scope for a set of IDL definitions. Modules allow you to group interface and other IDL type definitions into logical name spaces. When writing IDL definitions, always use modules to avoid possible name clashes.

The following example illustrates the use of modules in IDL:

```
// IDL
module finance {
   interface account {
      ...
   };
};
```

The interface `account` is *scoped* within the module `finance`. IDL definitions are available directly within the scope in which they are defined. In other naming scopes, you must use the scoping operator `::` to access these definitions. For example, the fully scoped name of interface `account` is `finance::account`.

IDL modules can be *reopened*. For example, a module declaration can appear several times in a single IDL specification if each declaration contains different data types. In most IDL specifications, this feature of modules is not required.

Defining IDL Interfaces

An IDL interface describes the functions that an object supports in a distributed application. Interface definitions provide all of the information that clients need to access the object across a network.

Consider the example of an interface which describes objects that implement bank accounts in a distributed application.

The IDL interface definition is as follows:

```
//IDL
module finance {
   interface account {
      // The account owner and balance.
      readonly attribute string owner;
      readonly attribute float balance;

      // Operations available on the account.
      void makeLodgement(in float amount,
         out float newBalance);
      void makeWithdrawal(in float amount,
         out float newBalance);
   };
};
```

The definition of interface `account` includes both *attributes* and *operations*. These are the main elements of any IDL interface definition.

IDL Attributes

Conceptually, IDL attributes correspond to variables that an object implements. Attributes indicate that these variables are available in an object and that clients can read or write their values.

In general, each attribute maps to a pair of functions in the programming language used to implement the object. These functions allow client applications to read or write the attribute values. However, if an attribute is preceded by the keyword `readonly`, clients can only read the attribute value.

For example, the `account` interface defines the attributes `balance` and `owner`. These attributes represent information about the account which the object implementation can set, but which client applications can only read.

IDL Operations

IDL operations define the format of functions, methods, or operations that clients use to access the functionality of an object. An IDL operation can take parameters and return a value, using any of the available IDL data types.

For example, the `account` interface defines the operations `makeLodgement()` and `makeWithdrawal()` as follows:

```
//IDL
module finance {
   interface account {
      // Operations available on the account.
      void makeLodgement(in float amount,
         out float newBalance);
      void makeWithdrawal(in float amount,
         out float newBalance);
      ...
   };
};
```

Each operation takes two parameters and has a `void` return type. The parameter definitions must specify the direction in which the parameter value is passed. The possible parameter passing modes are as follows:

`in`	The parameter is passed from the caller of the operation to the object.
`out`	The parameter is passed from the object to the caller.
`inout`	The parameter is passed in both directions.

Parameter passing modes clarify operation definitions and allow an IDL compiler to map operations accurately to a target programming language.

Raising Exceptions in IDL Operations

IDL operations can raise exceptions to indicate the occurrence of an error. CORBA defines two types of exceptions:

- *System exceptions*
  These are a set of standard exceptions defined by CORBA.
- *User-defined exceptions*
  These are exceptions that you define in your IDL specification.

All IDL operations can implicitly raise any of the CORBA system exceptions. No reference to system exceptions appears in an IDL specification. Refer to the *OrbixWeb Programmer's Reference* for a complete list of the CORBA system exceptions.

To specify that an operation can raise a user-defined exception, first define the exception structure and then add an IDL `raises` clause to the operation definition. For example, the operation `makeWithdrawal()` in interface `account` could raise an exception to indicate that the withdrawal has failed, as follows:

```
// IDL
module finance {
   interface account {
      exception WithdrawalFailure {
         string reason;
      };

      void makeWithdrawal(in float amount,
         out float newBalance)
         raises(WithdrawalFailure);
      ...
   };
};
```

An IDL exception is a data structure that contains member fields. In this example, the exception `WithdrawalFailure` includes a single member of type string.

The `raises` clause follows the definition of operation `makeWithdrawal()` to indicate that this operation can raise exception `WithdrawalFailure`. If an operation can raise more then one type of user-defined exception, include each exception identifier in the `raises` clause and separate the identifiers using commas.

Invocation Semantics for IDL Operations

By default, IDL operation calls are *synchronous.* This means that a client calls an operation and blocks until the object has processed the operation call and returned a value. The IDL keyword `oneway` allows you to modify these invocation semantics.

If you precede an operation definition with the keyword `oneway`, a client that calls the operation will not block while the object processes the call. For example, you could add a oneway operation to interface `account` that sends a notice to an `account` object, as follows:

```
module finance {
   interface account {
      oneway void notice(in string text);
      ...
   };
};
```

OrbixWeb does not guarantee that a oneway operation call will succeed; so if a oneway operation fails, a client may never know. There is only one circumstance in which OrbixWeb indicates failure of a oneway operation. If a oneway operation call fails *before* OrbixWeb transmits the call from the client address space, then OrbixWeb raises a system exception.

Note: A oneway operation cannot have any `out` or `inout` parameters and cannot return a value. In addition, a oneway operation cannot have an associated `raises` clause.

Passing Context Information to IDL Operations

CORBA context objects allow a client to map a set of identifiers to a set of string values. When defining an IDL operation, you can specify that the operation should receive the client mapping for particular identifiers as an implicit part of the operation call. To do this, add a `context` clause to the operation definition.

Consider the example of an `account` object, where each client maintains a set of identifiers, such as `sys_time` and `sys_location` that map to information that the operation `makeLodgement()` logs for each lodgement received. To ensure that this information is passed with every operation call, extend the definition of `makeLodgement()` as follows:

```
// IDL
module finance {
   interface account {
      void makeLodgement(in float amount,
         out float newBalance)
         context("sys_time", "sys_location");
      ...
   };
};
```

A `context` clause includes the identifiers for which the operation expects to receive mappings. IDL contexts are rarely used in practice.

Inheritance of IDL Interfaces

IDL supports inheritance of interfaces. An IDL interface can inherit all the elements of one or more other interfaces.

For example, the following IDL definition illustrates two interfaces, called `checkingAccount` and `savingsAccount`. Both of these inherit from an interface named `account`:

```
// IDL
module finance {
   interface account {
      ...
   };

   interface checkingAccount : account {
      readonly attribute overdraftLimit;
      boolean orderChequeBook ();
   };

   interface savingsAccount : account {
      float calculateInterest ();
   };
};
```

Interfaces `checkingAccount` and `savingsAccount` implicitly include all elements of interface `account`.

An object that implements `checkingAccount` can accept calls on any of the attributes and operations of this interface, and also on any of the elements of interface `account`. However, a `checkingAccount` object may provide different implementations of the elements of interface `account` to an object that implements `account` only.

The following IDL definition shows how to define an interface that inherits both `checkingAccount` and `savingsAccount`:

```
// IDL
module finance {
   interface account {
      ...
   };

   interface checkingAccount : account {
      ...
   };

   interface savingsAccount : account {
      ...
   };

   interface premiumAccount :
      checkingAccount, savingsAccount {
   };
};
```

Interface `premiumAccount` is an example of multiple inheritance in IDL. Figure 14 illustrates the inheritance hierarchy for this interface.

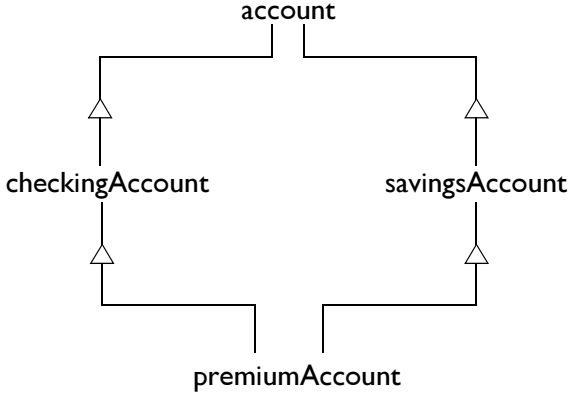

Figure 14: *Multiple Inheritance of IDL Interfaces*

If you define an interface that inherits from other interfaces containing a constant, type, or exception definition of the same name, you must fully scope that name when using the constant, type, or exception.

Note: An interface cannot inherit from other interfaces which include operations or attributes that have the same name.

The Object Interface Type

IDL includes the pre-defined interface `Object`, which all user-defined interfaces inherit implicitly. The operations defined in this interface are described in the *OrbixWeb Reference Guide.* While interface `Object` is never defined explicitly in your IDL specification, the operations of this interface are available through all your interface types. In addition, you can use `Object` as an attribute or operation parameter type to indicate that the attribute or operation accepts any interface type, for example:

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

It is not legal IDL syntax to explicitly inherit interface `Object`.

Forward Declaration of IDL Interfaces

In IDL, you must declare an IDL interface before you reference it. A forward declaration declares the name of an interface without defining it. This feature of IDL allows you to define interfaces that mutually reference each other.

For example, IDL interface `account` could include an attribute of IDL interface type `bank`, to indicate that an `account` stores a reference to a `bank` object. If the definition of interface `bank` follows the definition of interface `account`, you would make a forward declaration for the `bank` interface as follows:

```
// IDL
module finance {
   // Forward declaration of bank.
   interface bank;
   interface account {
      readonly attribute bank branch;
      ...
   };

   // Full definition of bank.
   interface bank {
      ...
   };
};
```

The syntax for a forward declaration is the keyword `interface` followed by the interface identifier.

Note: It is not possible to inherit from a forwardly declared interface. You can only inherit from an interface which has been fully specified.

The following IDL definition, for example, is not permitted:

```
//IDL
module finance{
   //Forward declaration of bank.
   interface bank;

   interface account Bigbank:bank{
      ...
   }
```

Overview of the IDL Data Types

In addition to IDL module, interface, and exception types, there are four main categories of data type in IDL:

- *Basic types*
- *Constructed types*
- *Template types*
- *Pseudo object types*

This section examines each IDL data type in turn, and describes how you can define new data type names, arrays, and constants in IDL.

IDL Basic Types

Table 1 lists the basic types supported in IDL.

IDL Type	Range of Values
`short`	$-2^{15}...2^{15}-1$ (16-bit)
`unsigned short`	$0...2^{16}-1$ (16-bit)
`long`	$-2^{31}...2^{31}-1$ (32-bit)
`unsigned long`	$0...2^{32}-1$ (32-bit)
`long long`	$-2^{63}...2^{63}-1$ (64-bit)
`unsigned long long`	$0...2^{63}-1$ (64-bit)
`float`	IEEE single-precision floating point numbers.
`double`	IEEE double-precision floating point numbers.
`char`	An 8-bit value.
`wchar`	A 16-bit value.
`boolean`	TRUE or FALSE.

Table 1: *The IDL Basic Types*

IDL Type	Range of Values
octet	An 8-bit value that is guaranteed not to undergo any conversion during transmission.
any	The any type allows the specification of values that can express an arbitrary IDL type.

Table 1: *The IDL Basic Types*

The any data type allows you to specify that an attribute value, an operation parameter, or an operation return value can contain an arbitrary type of value to be determined at runtime. Refer to "Type any" on page 347 for more details.

IDL Constructed Types

IDL provides three constructed data types:

- enum
- struct
- union

Enum

An enumerated type allows you to assign identifiers to the members of a set of values, for example:

```
// IDL
module finance {
   enum currency {pound, dollar, yen, franc};

   interface account {
      readonly attribute float 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 to indicate the currency associated with the attribute balance.

Struct

A struct data type allows you to package a set of named members of various types, for example:

```
// IDL
module finance {
   struct customerDetails {
      string name;
      short age;
   };

   interface bank {
      customerDetails getCustomerDetails(
         in string name);
         ...
   };
};
```

In this example, the struct `customerDetails` has two members: `name` and `age`. The operation `getCustomerDetails()` returns a struct of type `customerDetails` that includes values for the customer name and age.

Union

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

All IDL unions are *discriminated*. This means that they associate a label value with each member. The value of the label indicates which member of the union currently stores a value.

For example, consider the following IDL union definition:

```
// IDL
struct DateStructure {
   short Day;
   short Month;
   short Year;
};

union Date switch (short) {
   case 1: string stringFormat;;
   case 2: long digitalFormat;
   default: DateStructure structFormat;
};
```

The union type `Date` is discriminated by a short value. For example, if this short value is `1`, the union member `stringFormat` stores a date value as an IDL string. The default label associated with the member `structFormat` indicates that if the short value is not `1` or `2`, the `structFormat` member stores a date value as an IDL struct.

The type specified in parentheses after the `switch` keyword must be an `integer`, `char`, `boolean` or `enum` type and the value of each `case` label must be compatible with this type.

IDL Template Types

IDL provides two template types:

- `string`
- `sequence`

String

An IDL string represents a character string, where each character can take any value of the `char` basic type.

If the maximum length of an IDL string is specified in the string declaration, the string is *bounded*. Otherwise the string is *unbounded*.

The following example shows how to declare bounded and unbounded strings:

```
// IDL
module finance {
   interface bank {
      // A bounded string with maximum length 10.
      attribute string sortCode<10>;
      // An unbounded string.
      attribute string address;
      ...
   };
};
```

Sequence

In IDL, you can declare a sequence of any IDL data type or user-defined data type. An IDL sequence is similar to a one-dimensional array of elements.

An IDL sequence does not have a fixed length. If the sequence has a fixed maximum length, then the sequence is *bounded*. Otherwise, the sequence is *unbounded*.

For example, the following code shows how to declare bounded and unbounded sequences as members of an IDL struct:

```
// IDL
module finance {
   interface account {
      ...
   };

   struct limitedAccounts {
      string bankSortCode<10>;
      // Maximum length of sequence is 50.
      sequence<account, 50> accounts;
   };

   struct unlimitedAccounts {
      string bankSortCode<10>;
      // No maximum length of sequence.
      sequence<account> accounts;
   };
};
```

A sequence must be named by an IDL `typedef` declaration (described in "Defining Aliases and Constants" on page 86) before it can be used as the type of an IDL attribute or operation parameter. This is illustrated by the following code:

```
// IDL
module finance {
   typedef sequence<string> customerSeq;

   interface bank {
      void getCustomerList(out customerSeq names);
      ...
   };
};
```

Arrays

In IDL, you can declare an array of any IDL data type. IDL arrays can be multidimensional and always have a fixed size. For example, you can define an IDL struct with an array member as follows:

```
// IDL
module finance {
   interface account {
      ...
   };

   struct customerAccountInfo {
      string name;
      account accounts[3];
   };

   interface bank {
      getCustomerAccountInfo (in string name,
         out customerAccountInfo accounts);
      ...
   };
};
```

In this example, struct `customerAccountInfo` provides access to an array of `account` objects for a bank customer, where each customer can have a maximum of three accounts.

As with sequences, an array must be named by an IDL `typedef` declaration before it can be used as the type of an IDL attribute or operation parameter. The following code illustrates this:

```
// IDL
module finance {
   interface account {
      ...
   };
   typedef account accountArray[100];

   interface bank {
      readonly attribute accountArray accounts;
      ...
   };
};
```

Note: Arrays are a less flexible data type than an IDL sequence, because an array always has a fixed length. An IDL sequence always has a variable length, although it may have an associated maximum length value.

IDL Pseudo-Object Types

CORBA defines a set of pseudo-object types that ORB implementations use when mapping IDL to some programming languages. 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:

- `NamedValue`
- `Principal`
- `TypeCode`

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

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

This statement indicates to the IDL compiler that types `NamedValue`, `Principal`, and `TypeCode` may be used. The file `orb.idl` does not actually exist in your system. Do not name any of your IDL files `orb.idl`.

For more information on these types, refer to "IDL to Java Mapping" on page 91, and to the *OrbixWeb Reference Guide*.

Defining Aliases and Constants

IDL allows you to define *aliases* (new data type names) and constants. This section describes how to use these IDL features.

Using Typedef to Create Aliases

The `typedef` keyword allows you define a more meaningful or simple name for an IDL type. The following IDL provides a simple example of using this keyword:

```
// IDL
module finance {
   interface account {
      ...
   };

   typedef account standardAccount;
};
```

The identifier `standardAccount` can act as an alias for type `account` in subsequent IDL definitions. CORBA does not specify whether the identifiers `account` and `standardAccount` represent distinct IDL data types in this example.

Constants

IDL allows you to specify constant data values using one of several basic data types. Appendix A, IDL Reference in the *OrbixWeb Programmer's Reference* indicates which data types you can use to define constants.

To declare a constant, use the IDL keyword `const`, for example:

```
// IDL
module finance {
   interface bank {
      const long MaxAccounts = 10000;
      const float factor = (10.0 - 6.5) * 3.91;
      ...
   };
};
```

The value of an IDL constant cannot change. You can define a constant at any level of scope in your IDL specification.

6

IDL to Java Mapping

This chapter describes OrbixWeb's mapping of IDL to Java, using the OrbixWeb IDL to Java compiler. OrbixWeb's implementation of the IDL to Java mapping conforms with version 1.1 of the standard OMG IDL/Java Language Mapping specification.[1] The chapter explains the rules used to convert IDL definitions into Java source code, as well as how to use the generated Java constructs.

An IDL definition is used to specify the interface for an object: This interface must then be implemented using an appropriate programming language. To allow implementation of interfaces in OrbixWeb, the IDL interfaces specified are mapped to Java, using the OrbixWeb IDL to Java compiler. This compilation produces a set of classes that allow the client to invoke operations on a remote object as if it were located on the same machine.

This chapter is designed to illustrate the fundamentals of the IDL to Java mapping, and to serve as a reference for more detailed technical information required when writing applications.

1. The *IDL/Java Language Mapping* specification is available from the OMG web site at `http://www.omg.org`

Overview of IDL to Java Mapping

The principal elements of the IDL to Java mapping are outlined as follows:

Basic Types

Basic types in IDL are mapped to the most closely corresponding Java type. All mapped basic types have *holder* classes which support parameter passing modes. Refer to "Mapping for Basic Data Types" on page 92.

Mapping for Modules

An IDL *module* is mapped to a Java package of the same name. Scoped names are used for types defined in interfaces within a module. Refer to "Mapping for Modules" on page 94 for details.

Mapping for Interfaces and Operation Parameters

IDL *interfaces* are mapped to Java interfaces and classes which provide client-side and server-side support. Provision is made for two approaches to interface implementation: the *TIE* and *Implbase* approaches.

Attributes within IDL interfaces are mapped to a pair of overloaded methods allowing the attribute value to be set and retrieved.

Operations within IDL interfaces are mapped to Java methods of the same name in the corresponding Java interface.

Helper classes are generated by the IDL compiler. These contain a number of static methods for type manipulation. Refer to "Helper Classes for Type Manipulation" on page 97.

Holder classes are generated by the IDL compiler for all user-defined types to implement parameter-passing modes in Java. Holder classes are needed because IDL `inout` and `out` parameters do not map directly into the Java parameter passing mechanism. Holder classes for the basic types are available in the `org.omg.CORBA` package. Refer to "Holder Classes and Parameter Passing" on page 100.

Mapping for Constructed Types

Constructed types map to a Java `final` class, containing methods and data members appropriate to the mapped type. For a full description of mapping for `enum`, `struct`, and `union` types, refer to "Mapping for Constructed Types" on page 117.

Mapping for Strings

IDL *strings*, both bounded and unbounded, map to the Java type `String`. OrbixWeb performs bounds checking for `String` parameter values passed as bounded strings to IDL operations. Refer to "Mapping for Strings" on page 123.

Mapping for Sequences and Arrays

IDL *sequences,* both bounded and unbounded, map to Java arrays of the same name. OrbixWeb performs bounds checking for bounded sequences. Helper and holder classes are generated for mapped IDL sequences. Refer to "Mapping for Sequences" on page 125.

IDL *arrays* map directly to Java arrays of the same name. OrbixWeb performs the bounds checking, since Java arrays are not bounded. Refer to "Mapping for Arrays" on page 126.

Mapping for Constants

Constants map to `public static final` fields in a corresponding Java interface. If the constant is not defined in an interface, the mapping first generates a public interface with the same name as the constant. Refer to "Mapping for Constants" on page 127.

Mapping for Typedefs

Typedefs are mapped to the corresponding Java mapping for the original IDL type. A helper class is generated for the declared type. The IDL to Java mapping for constants and *typedefs* is described in "Mapping for Typedefs" on page 129.

Mapping for Exceptions

IDL *standard system exceptions* are mapped to Java `final` classes which extend `org.omg.CORBA.SystemException` and provide access to IDL exception code. IDL *user-defined exception* types map to a `final` class which derives from `org.omg.CORBA.UserException`. User-defined exceptions have helper and holder classes generated. Refer to "Mapping for Exception Types" on page 129.

Mapping for Basic Data Types

The IDL basic data types are mapped to corresponding Java types as shown in Table 1.

IDL	JAVA	Exceptions
short	short	
long	int	
unsigned short	short	
unsigned long	int	
long long	long	
unsigned long long	long	
float	float	
double	double	
char	char	CORBA::DATA_CONVERSION
wchar	char	CORBA::DATA_CONVERSION
string	java.lang.String	CORBA::MARSHAL CORBA::DATA_CONVERSION
wstring	java.lang.String	CORBA::MARSHAL CORBA::DATA_CONVERSION
boolean	boolean	
octet	byte	
any	org.omg.CORBA.Any	

Table 1: *Mapping for Basic Types*

You should note the following features of the IDL to Java mapping for basic types:

- **Holder Classes for Parameter Passing**

  All IDL basic types have holder classes available in the `org.omg.CORBA` package to provide support for the `out` and `inout` parameter passing modes. For more details on holder classes refer to "Holder Classes and Parameter Passing" on page 100.

- **IDL Long Maps to Java Int**

  The 32-bit IDL `long` is mapped to the 32-bit Java `int`.

- **IDL Unsigned Types Map to Signed Java Types**

  Java does not support unsigned data types. All `unsigned` IDL types are mapped to the corresponding signed Java types. You should ensure that large unsigned IDL type values are handled correctly as negative integers in Java.

- **IDL Chars and Java Chars**

  IDL `chars` are based on the 8-bit character set for ISO 8859.1. Java `chars` come from the 16-bit UNICODE character set. Consequently IDL `chars` only represent a small subset of Java `chars`. On marshalling, if a `char` has a value outside the range defined by the character set, a `CORBA::DATA_CONVERSION` exception is thrown. The 16-bit IDL `wchar` represents the full range of Java `chars`, and maps to the Java primitive type `char`.

- **IDL Strings**

  IDL `string` types map to the Java type `String`. On marshalling, range checking for characters and bounds checking of the string is performed. Character range violations raise a `CORBA::DATA_CONVERSION` exception; bounds violations raise a `CORBA::MARSHAL` exception. IDL `wstring` types, both bounded and unbounded, also map to the Java type `String`.

- **Booleans**

  The IDL `boolean` type constants `TRUE` and `FALSE` map to the Java `boolean` type literals `true` and `false`.

- **Type any**

  The mapping for type `any` is described in full in "Type any" on page 347.

Mapping for Modules

An IDL module is mapped to a Java package of the same name. All IDL type declarations within the module are mapped to a corresponding Java class or interface declaration within the generated package. IDL declarations *not* enclosed in any modules are mapped into the Java global scope. The use of modules is recommended.

Scoped Names

All types defined inside an IDL module are mapped within a Java package with the same name as that module. For example, if an interface named `bank` is defined inside the module `IDLDemo`, then the Java interface for `bank` is scoped as `IDLDemo.bank`.

Similarly, any type defined inside an interface is scoped first by the module name, if defined, and then by a package named `<type>Package`, where `<type>` is the interface name. Therefore, if `bank` defines a structure called `Details`, the corresponding class is scoped as `IDLDemo.bankPackage.Details`.

IDL types which are not defined inside either a module or an interface are not included in a Java package. This creates the potential for naming collisions with other globally defined Java types. To avoid the generation of such naming collisions, always define your IDL within modules. Alternatively, use the `-jP` compiler option, which specifies a package prefix that is added to generated types. This makes it possible to use globally defined IDL types within a package scope.

Refer to the *OrbixWeb Programmer's Reference* for more details on the use of compiler options.

The CORBA Module

The objects and data types pre-defined in CORBA are logically defined within an IDL module called `CORBA`. IDL maps the CORBA module to a Java package called `org.omg.CORBA`. In line with this mapping, the OMG keyword `Object` maps to `org.omg.CORBA.Object`.

In OrbixWeb, the `org.omg.CORBA` set of classes represents the OMG standard abstract runtime. The actual implementation of the OrbixWeb ORB resides in the `IE.Iona.OrbixWeb` package.

Mapping for Interfaces

An IDL interface maps to a public Java interface of the same name, and a number of other generated Java constructs. This discussion focuses on the client-side and server-side mapping, and on *helper* and *holder* classes. These classes have roles on both the client-side and the server-side.

IDL interface definitions are compiled by the IDL to Java compiler. The following Java constructs are generated, where `<type>` represents a user-defined interface name:

Generated Files	Description	Side
`<type>.java`	Java Reference interface	client
`_<type>Stub.java`	Java Stub class	client
`_<type>Skeleton.java`	Java Skeleton class	server
`_<type>ImplBase.java`	ImplBase class	server
`_tie_<type>.java`	TIE class	server
`_<type>Operations.java`	Java interface (used with TIE class)	server
`<type>Helper.java`	Java Helper class	client/server
`<type>Holder.java`	Java Holder class	client/server
`<type>Package`	Java package.	client/server

Note: The classes `_tie_<type>.java` and `_<type>Operations.java` are specific to OrbixWeb. To generate files defined by CORBA only, use the `-jOMG` IDL Compiler switch.

This section uses the IDL interface `account` to show how an IDL interface is mapped to Java:

```
// IDL
module bank_demo;
interface account {
   readonly attribute float balance;

   void makeLodgement(in float sum);
   void makeWithdrawal(in float sum);
};
```

Client Mapping

The OrbixWeb client provides proxy functionality for the IDL interface. The IDL compiler generates the following client-side Java constructs for each IDL interface:

- *Java Reference interface*
- *Java Stub class*
- *Java Helper class*
- *Java Holder class*

Java Reference Interface

A *Java Reference Interface* type has the naming format `<type>.java`. It defines the client view of the IDL interface, listing the methods that a client can call on objects which implement the IDL type. The interface extends the base `org.omg.CORBA.Object` interface.

The following Java Reference interface for the IDL interface `account` illustrates the Java mapping for IDL attributes and operations:

```
// Java generated by the OrbixWeb IDL compiler

package bank_demo;

public interface account
           extends org.omg.CORBA.Object {
   public float balance();
   public void makeLodgement(float sum);
   public void makeWithdrawal(float sum);
}
```

The read-only attribute `balance` maps to a single Java method, since there is no requirement for setting its value.

The IDL operations `makeLodgement` and `makeWithdrawal` map to methods of the same name in the corresponding Java interface.

Java Stub Class

The *Java Stub* class generated by the IDL compiler implements the Java interface and provides the functionality to allow client invocations to be forwarded to the server. This class has a naming format of `_<type>Stub.java`. This generated class is used internally by OrbixWeb and you do not need to understand how it works.

Java Helper classes and Java Holder classes are discussed in the following two sections.

Helper Classes for Type Manipulation

A *Java Helper* class is also generated by the Java mapping. Helper classes contain methods that allow IDL types to be manipulated in various ways. The IDL-to-Java compiler generates helper classes for all IDL user-defined types. The naming format for helper classes is `<type>Helper`, where `<type>` is the name of an IDL user-defined type.

Helper classes include methods that support insertion and extraction of the `account` object into and from Java `Any` types. Interface Helper classes also have static class methods for `narrow()` and `bind()`. The `narrow()` method takes an `org.omg.CORBA.Object` type as an argument, and returns an object reference of the same type as the class. The `bind()`[2] method may be used to create a *proxy* for an object that implements the IDL interface. A proxy object is a client-side representative for a remote object. Operations invoked on the proxy result in requests being sent to the target object.

The following code illustrates the Java Helper class generated from the IDL `account` interface:

```
// in file accountHelper.java
// Java generated by the OrbixWeb IDL compiler
//
import org.omg.CORBA.Any;
import org.omg.CORBA.Object;
import org.omg.CORBA.TypeCode;
```

2. `bind()` is a feature specific to OrbixWeb. If you wish to use only those features defined in the CORBA specification, you should compile your IDL using the `-jOMG` switch.

```
import org.omg.CORBA.portable.OutputStream;
import org.omg.CORBA.portable.InputStream;

public class accountHelper {

1     public static void insert (Any any, account value) {
         ...
      }
      public static account extract (org.omg.CORAny any) {
         ...
      }
2     public static TypeCode type () {
         ...
      }
3     public static String id () {
         ...
      }
4     public static account read (InputStream _stream) {
         ...
      }
      public static void write (OutputStream _stream, account value){
         ...
      }
5     public static final account bind() {
         ...
      }
      public static final account bind(String markerServer) {
         ...
      }
      public static final account bind(org.omg.CORBA.ORB orb) {
         ...
      }
      public static final account bind
         (String markerServer, String host){
         ...
      }

      public static final account bind
         (String markerServer, org.omg.CORBA.ORB orb){
         ...
      }
```

```
        public static final account bind
           (String markerServer, String host, org.omg.CORBA.ORB orb){
           ...
        }
6       public static account narrow (Object _obj) {
           ...
        }
     }
```

These methods provided by helper classes are described as follows:

1. The `insert()` and `extract()` methods allow for IDL interface types to be passed as a parameter of IDL type `any`. Refer to "Type any" on page 347 for further information on this topic.
2. The `type()` method returns a `TypeCode` for a specified interface.

   `TypeCodes` allow runtime querying of type information for an `Any` type. They can also be used for interrogating the Interface Repository. Refer to Chapter 19, "TypeCode" for more details.
3. The `id()` method is used to retrieve the Repository ID for the object.
4. The `read()` and `write()` methods allow the type to be written to and from a stream.
5. The `bind()` method provides an alternative to using the Naming Service, and is a feature specific to OrbixWeb.

   `bind()` locates a specified object and creates a proxy for it in the client's address space. Overloaded methods within `bind()` allow a variety of parameters to be passed. Refer to "The bind() Method" on page 204 for a description to the parameters to `bind()`.

   The Naming Service is the preferred method for locating objects in servers, but some applications may benefit from using the `bind()` method.

   Using the Bind() Method

   A client wishing to use the IDL interface should bind an object of the Java class type to the target implementation object in the server, assigning the result to the Java Reference interface type.

For example, a client could bind to an account implementation object by calling the bind() static method on the Java accountHelper class as follows:

```
// Java
account aRef;
aRef = accountHelper.bind();
```

This returns a proxy object which can be accessed using the methods defined in the account interface.

6. The narrow() method allows an interface to be safely cast to a derived interface. For example, it allows an org.omg.CORBA.Object to be narrowed to the object reference of a more specific type. For IDL-defined objects, you must use narrow() rather than the normal Java cast operation. Failure of the method raises a CORBA::BAD_PARAM exception.

   Refer to "Mapping for Derived Interfaces" on page 112 for further information on narrowing object references.

Holder Classes and Parameter Passing

IDL in parameters always map directly to the corresponding Java type. This mapping is possible because in parameters are always passed by value, and Java supports by value passing of all types. Similarly, IDL return values always map directly to the corresponding Java type.

IDL inout and out parameters, however, must be passed by reference, because they may be modified during an operation call, and do not map directly into the Java parameter passing mechanism. In the IDL to Java mapping, IDL inout and out parameters are mapped to *Java Holder* classes. Holder classes simulate passing by reference. The client supplies an instance of the appropriate Java holder class passed by value, for each IDL out or inout parameter. The contents of the holder instance are modified by the call and the client uses the contents when the call returns.

There are two categories of holder classes:

- Holders for *basic* types.
- Holders for *user-defined* types.

Holders for Basic Types

Holder classes for *basic* Java types and the Java `string` type, are available in the package `org.omg.CORBA`. The name format used is `<type>Holder`, where `<type>` is the name of a basic Java type, with initial capital letter, for example `IntHolder`.

An example of the implementation for `IntHolder` follows:

```
// Java
package org.omg.CORBA;
public class IntHolder {
1       public int value;
        public IntHolder () {}
2       public IntHolder (int value) {
           this.value = value;
        }
}
```

1. The holder class stores an `int` value as a member variable.
2. The value can be initialized by the constructor and accessed directly. The holder class simulates passing by reference to method invocations and so facilitates the modification of an `int`, which would not be possible if the `int` were passed directly.

Holders for User-Defined Types

Holder classes for *user-defined* types, including IDL interface types, are generated by the Java mapping. The name format is `<type>Holder`. For example, given an IDL interface `account`, the following `Holder` class is generated:

```
// in file accountHolder.java
// Java generated by the OrbixWeb IDL compiler
//
public final class accountHolder {
1         public account value;
          public accountHolder() {};
          public accountHolder(account value) {
             this.value = value;
          }
          ...
}
```

1. The holder class stores an `account` value as a member variable, which can be initialized by the constructor and accessed directly.

Invoking an Operation using Holder Classes

When using holder classes to pass `inout` and `out` parameters, the following rules apply:

- The client programmer must supply an instance of the appropriate holder Java class that is passed, *by value*, for each IDL `out` or `inout` parameter.

  The contents of the holder instance are modified by the call, and the client then uses the contents after the call returns.
- For the `inout` parameter, the client *must* initialize the holder with a valid value. The operation can examine the value supplied by the client and may change the value if it wishes. The final value at the end of the operation (changed or not) is returned to the client.
- For the `out` parameter: the client does not need to initialize the holder with a value, as any value in the holder is ignored. The operation should *not* use the initial value in the holder and *must* supply a valid value to be returned to the client.

To illustrate the use of holder types, consider the following IDL definition:

```
// IDL

void newAccount
   (in string name, out account acc, out string accID)
```

The IDL compiler maps this operation to a method of Java interface `bank` as follows:

```
// In package bank_demo.bank,

public void newAccount(String name, bank_demo.accountHolder acc,
                                    org.omg.CORBA.StringHolder accID);
```

This method returns an object reference to the interface `account` and a `string` value of a variable `accID`, which is an account number automatically generated by the server object. Holder classes are generated for the `out` return values to allow the server to pass back new values to the client.

The holder class accountHolder stores a value member variable of type Account, which may be modified during the operation call.

```
//
// Java generated by the OrbixWeb IDL compiler
// accountHolder.java
package bank_demo
public final class accountHolder {
1       public bank_demo.account value;
        public accountHolder() {}
2       public accountHolder(bank_demo.account value) {
        ...
        }
}
```

1. The value variable is of type account.
2. value can be initialized by a constructor and accessed directly. The holder class simulates passing by reference to method calls and so allows value to be changed. This would not be possible if value was passed directly.

A client application can be coded as follows:

```
// Java
// In file javaclient1.java.
import org.omg.CORBA.SystemException;

public class javaclient1{
   public static void main (String args[]) {
      bank bRef = null;
      account aRef = null;
      accountHolder aHolder =
         new accountHolder ();
      float f = (float) 0.0;

      try {
         // Bind to any bank object
         // in BankSrv server.
         bRef = bankHelper.bind (":BankSrv");
```

```
            // Obtain a new bank account.
            bRef.newAccount ("Joe", aHolder);
         }
         catch (SystemException se) {
            System.out.println (
               "Unexpected exception on bind");
            System.out.println (se.toString ());
            System.exit(1);
         }

         // Retrieve value from Holder object.
         aRef = aHolder.value;

         try {
            // Invoke operations on account.
            aRef.makeLodgement ((float)56.90);
            f = aRef.balance();
            System.out.println ("Current balance is + f);
         }
         catch (SystemException se) {
            System.out.println (
               "Unexpected exception"
               + " on makeLodgement or balance");
            System.out.println (se.toString ());
            System.exit(1);
         }
   }
```

In the server, the implementation of method `newAccount()` receives the `Holder` object for type `account` and may manipulate the value field as required. For example, in this case the `newAccount()` method can instantiate a new `account` implementation object as follows:

```
// Java
// In class bankImplementation.
public void newAccount
   (String name,bank_demo.accountHolder acc) {
      accountImplementation accImpl =
         new accountImplementation (0, name);

      acc.value = new _tie_account (accImpl);
      ...
}
```

Note: If the account parameter is labelled inout in the IDL definition, the value member of the Holder class needs to be instantiated before calling the newAccount() operation.

Server Implementation Mapping

The Java mapping generates four classes to support server implementation in OrbixWeb. The following files are generated:

- A *Java Skeleton* class, with the name format _<type>Skeleton.java, used internally by OrbixWeb to dispatch incoming server requests to implementation objects. You do not need to know the details of this class.
- An abstract *Java ImplBase* class, with the name format _<type>ImplBase.java, which allows server-side developers to implement interfaces using the ImplBase approach.
- A *Java TIE* class, with the name format _tie_<type>.java, that allows server side developers to implement interfaces using delegation (the TIE approach[3]).
- A *Java Operations* interface, with the name format _<type>Operations, that is used in the TIE approach to map the attributes and operations of the IDL definition to Java methods. This class is specific to OrbixWeb, and is used to support implementation using the TIE approach.

3. The TIE Approach is specific to OrbixWeb. If you wish to use only those features defined in the CORBA specification, you should compile the IDL using the -jOMG switch.

Approaches to Interface Implementation

OrbixWeb supports two approaches to the implementation of IDL interfaces in Java applications:

- The ImplBase approach.
- The TIE approach.

This section discusses the Java types generated to enable each implementation method.

Both approaches to interface implementation share the common requirement that you *must* create a Java implementation class. This class must fully implement methods corresponding to the attributes and operations of the IDL interface.

The ImplBase Approach to Implementation

To support the ImplBase approach, the IDL compiler generates an abstract Java class from each IDL interface definition. This abstract class is named by adding `ImplBase` to the IDL interface name, prefixed by an underscore. For example, the compiler generates class `_accountImplBase` from the definition of interface `account`.

To implement an IDL interface using the ImplBase approach, you must create a Java class which extends the corresponding ImplBase class and implements the abstract methods.

For example, given the IDL definition for interface `account`, the compiler generates the abstract class `_accountImplBase` as follows[4]:

```
//
// Java generated by the OrbixWeb IDL compiler
//  _accountImplBase.java
//
import IE.Iona.OrbixWeb.Features.LoaderClass;

public abstract class _accountImplBase
   extends _accountSkeleton implements account {
```

4. In this code example, imports such as the `marker` and `loader` constructors are specific to OrbixWeb. To generate code which uses only those features defined in the CORBA specification, compile the IDL using the `-jOMG` switch.

```
   public _accountImplBase() {
      ...
   }
   public _accountImplBase(String marker) {
      ...
   }
   public _accountImplBase (LoaderClass loader){
      ...
   }
   public _accountImplBase(String marker,
                           LoaderClass loader) {
      ...
   }
```

A sample class, which implements the IDL interface `account` could contain code similar to the following:

```
// Java Implementation Class

class accountImplementation
   extends _accountImplBase {

   public accountImplementation(){
      ...
   }
   public float balance() {
      ...
   }
   public String get_name()
      ...
   }
   public void makeLodgement(float sum){
      ...
   }
   public void makeWithdrawal(float sum,
      ...
   }
...
}
```

Once the IDL interface has been implemented using the ImplBase approach, the server application should simply instantiate one or more objects of the implementation class. These objects can then handle client requests through the IDL interface in question.

The TIE Approach to Implementation

The IDL compiler generates a Java interface which defines the minimum set of methods which you must supply in order to implement an IDL interface using the TIE approach. The TIE approach is specific to OrbixWeb. To use only those features defined in the CORBA specification, compile your IDL with the `-jOMG` switch.

The name of this Java interface has the following format:

```
_<type>Operations
```

For example, given the IDL definition of type `account`, the IDL compiler generates the Java interface `_accountOperations` as follows:

```
// Java generated by the OrbixWeb IDL compiler

public interface _accountOperations {
   public float balance();
   public void makeLodgement(float sum);
   public void makeWithdrawal(float sum)
}
```

To support the TIE approach to implementation, the IDL compiler generates a non-abstract Java class from each IDL interface definition. This class is named by appending the IDL interface name to the string `_tie_`. For example, the compiler generates class `_tie_account` from the definition of interface `account`:

```
// Java generated by the OrbixWeb IDL compiler
// in file _tie_account.java
import IE.Iona.OrbixWeb._OrbixWeb;
import IE.Iona.OrbixWeb.Features.LoaderClass;

public class _tie_account extends _accountSkeleton
   implements account {

   public _tie_account(_accountOperations impl) {
      ...
   }
```

```
        public _tie_account
           (_accountOperations impl, String marker) {
           ...
        }
        public _tie_account
           (_accountOperations impl, LoaderClass loader) {
           ...
        }
        public _tie_account
           (_accountOperations impl, String marker,
              LoaderClass loader) {
           ...
        }
        public float balance(){
           ...
        }
        public String get_name()
           ...
        }
        public void makeLodgement(float sum) {
           ...
        }
        public void makeWithdrawal(float sum) {
           ...
        }
        public java.lang.Object _deref() {
           ...
        }
           ...
        }
}
```

When implementing an IDL interface using the TIE approach, the Java implementation class must directly implement the `Operations` interface. Unlike the ImplBase approach, the implementation class is not required to inherit from any other Java class. The TIE approach is therefore the recommended approach for Java programming, because of Java's restriction to single inheritance. Refer to Chapter 7, "Using and Implementing IDL Interfaces" for a detailed discussion of the TIE and ImplBase approaches.

The class accountImplementation could be outlined using the TIE approach as follows:

```
// Java generated by the OrbixWeb IDL compiler
// in file accountImplementation.java
class accountImplementation implements _accountOperations {

   public accountImplementation() {}
   public float balance() {
      ...
   }

   public float get_name() {
      ...
   }

   public void makeLodgement(float sum) {
      ...
   }
   public void makeWithdrawal(float sum) {
      ...
   }
}
```

When you have created an implementation class which implements the required Operations interface, the server application should instantiate one or more objects of this type. For each implementation object, the server should also instantiate an object of the corresponding TIE class, passing the implementation object as a parameter to the TIE constructor, as in the following example:

```
accountImpl = new accountImplementation();
account x = new _tie_account(accountImpl);
```

Each TIE object stores a reference to a single implementation object. Client operation invocations through the IDL interface are routed to the appropriate TIE object which then delegates the call to the appropriate method in its implementation object.

Object References

When an interface type is used in IDL, this denotes an object reference. For example, consider the IDL operation newAccount() defined as follows:

```
// IDL
interface account;
```

```
interface bank {
   account newAccount(in string name);
};
```

The return type of `newAccount()` is an object reference. An object reference maps to a Java interface of the same name. This interface allows IDL operations to be invoked on the object reference with normal Java method invocation syntax. For example, the `newAccount()` operation could be invoked as follows:

```
// Java
...
bank b;
account a;
...
b = bankHelper.bind ();
a = b.newAccount ("Chris");
a.makeLodgement ((float) 10.0);
...
```

The server implementation of operation `newAccount()` creates an `account` implementation object, stores a reference to this object, and returns the object reference to the client. For example, using the ImplBase approach and an implementation class named `accountImplementation`, you could do the following:

```
class bankImplementation
   extends _bankImplBase {

   public account m_acc;

   public bankImplementation () {
      m_acc=null;
   }
   public account newAccount(String name) {
      account a = null;

   try {
      a = new accountImplementation(0,name);
   }
   ....
   m_acc = a;
   return a;
   }
}
```

Similarly, you could use the TIE approach as follows:

```
class bankImplementation
   implements _bankOperations {

   public account m_acc;
   public bankImplementation () {
      m_acc=null;
   }
   public account newAccount(String name) {
      account a = null;
   try {
      a = new _tie_account(
      new accountImplementation(0,name));
   }
   ...
   m_acc = a;
   return a;
   }
}
```

If the operation `newAccount()` returned the `account` object reference as an `inout` or `out` parameter value, you need to pass the generated class `accountHolder` to the `newAccount()` Java method. `accountHolder` is a class which can contain an `account` object reference value.

Mapping for Derived Interfaces

This section describes the mapping for interfaces that inherit from other interfaces. Additional details of this mapping are provided in "Using Inheritance of IDL Interfaces" on page 305.

IDL interfaces support both single and multiple inheritance. On the client side, the OrbixWeb IDL compiler maps IDL interfaces to Java interfaces, which also support single and multiple inheritance, and generates Java classes which implement proxy functionality for these interfaces. Inherited interfaces in IDL are mapped to extended interfaces in Java; the inheritance hierarchy of the Java interfaces matches that of the original IDL interfaces.

Consider the following example:

```
// IDL
interface account {
   readonly attribute float balance;
   attribute String name;

   void makeLodgement(in float sum);
   void makeWithdrawal(in float sum);
};

interface checkingAccount : account {
   void overdraftLimit(in float limit);
};
```

The corresponding Java interface for type `checkingAccount` is:

```
// Java generated by the OrbixWeb IDL compiler
//
public interface checkingAccount extends account {
   public void setOverdraftLimit(float limit) ;
```

The corresponding Java stub class implements all methods for both `account` and `checkingAccount`. The generated class looks like this:

```
// Java generated by the OrbixWeb IDL compiler

import org.omg.CORBA.portable.ObjectImpl;

public class _checkingAccountStub
   extends ObjectImpl implements checkingAccount {

   public _checkingAccountStub () {}

   public void overdraftLimit(float limit){
      ...
   }
   public float balance() {
      ...
   }
   public float get_name() {
      ...
   }
```

```
        public void makeLodgement(float sum) {
           ...
        }
        public void makeWithdrawal(float sum) {
           ...
        }
        public String[] _ids() {
           ...
        }
    }
```

As expected, Java code you write which uses the `checkingAccount` interface can call the inherited `makeLodgement()` method:

```
// Java
checkingAccount checkingAc;

// Code for binding checkingAc {
   ...

   checkingAc.makeLodgement((float)90.97);
      ...
}
```

Assignments from a derived to a base class object reference are allowed, for example:

```
// Java
account ac = checkingAc;
```

Normal or cast assignments in the opposite direction—from a base class object reference to a derived class object reference—are not generally allowed. Use the `narrow()` method to bypass this restriction where it is safe to do so, as described in "Narrowing Object References" on page 116.

On the server side, the IDL compiler generates a Java `Operations` interface for each IDL interface. The generated Java interface defines the minimum set of implementation methods required for the IDL interface when using the TIE approach to implementation. The inheritance hierarchy of generated `Operations` interfaces matches that of the original IDL interfaces.

To implement an IDL interface which derives from another, define an implementation class which extends the ImplBase class for the required interface and implements all the methods defined in the ImplBase class.

For example, given the IDL definition of `account` and `checkingAccount`, a `checkingAccount` implementation class appears as follows:

```
// Java
// In file checkingAccountImplementation.java.
...
import org.omg.CORBA.FloatHolder;

public class checkingAccountImplementation
   extends _checkingAccountImplBase {
      public checkingAccountImplementation() {
         ...
      }
   public float balance() {
      ...
   }
   public float get_name() {
      ...
   }
   public void makeLodgement(float sum) {
      ...
   }
   public void makeWithdrawal(float sum) {
      ...
   }
   public void overdraftLimit(float limit) {
      ...
   }
}
```

Using the TIE approach, the implementation class should implement the generated `Operations` interface for the relevant IDL type. The implementation class must implement each method defined in the `Operations` interface and all interfaces from which it inherits. However, you can achieve this using an inheritance hierarchy of implementation classes, because the TIE approach, unlike the ImplBase approach, imposes no implicit inheritance requirements on such classes.

For example, if the IDL type `account` is implemented by class `accountImplementation`, using the TIE approach, you might implement IDL interface `checkingAccount` with type `checkingAccountImplementation` as follows:

```
// Java
// In file checkingAccountImplementation.java
...
public class checkingAccountImplementation
   extends accountImplementation,
   implements _checkingAccountOperations {

   public checkingAccountImplementation() {}

   public void overdraftLimit (float limit) {
      ...
   }
}
```

Narrowing Object References

In the `checkingAccount` example above, if you know that a reference of type `account` actually references an object which implements interface `checkingAccount`, you can *narrow* the object reference to a `checkingAccount` reference.

To narrow an object reference, use the `narrow()` method, defined as a `static` method in each generated Interface helper class.

```
// Java Generated by OrbixWeb IDL Compiler

import org.omg.CORBA.Object;

public class checkingAccountHelper {
   ...
   public static final checkingAccount narrow(Object src) {
      ...
   }
   ...
}
```

You can call the narrowed object reference as follows:

```
// Java
account a;
...
a = getCheckingAccountObject();
...
checkingAccount c;

// Narrow a to be a checkingAccount.
c = checkingAccountHelper.narrow(a);
```

If the parameter passed to `THelper.narrow()` is not of class `T` or one of its derived classes, `T.narrow()` raises the `CORBA.BAD_PARAM` exception.

Mapping for Constructed Types

The following sections describe the IDL to Java mapping for the `enum`, `struct` and `union` constructed types.

Enums

An `enum` declaration creates a correspondence between a set of integer values and a set of named values.

The following IDL definition illustrates an `enum` construct:

```
//IDL

enum Fruit { apple, orange};
```

An `enum` is mapped to Java according to the rules described for the mapping of the `enum Fruit` in the following example.

```
// Java generated by the OrbixWeb IDL compiler

public final class Fruit {
    public static final int _apple = 0;
    public static final Fruit apple = new Fruit(_apple);
    public static final int _orange = 1;
    public static final Fruit orange = new Fruit(_orange);
```

```
4         public int value () {
            ...
          }
5         public static Fruit from_int (int value) {
            ...
          }
      }
```

1. The IDL `enum` called `Fruit` maps to a Java `final` class of the same name.
2. The `enum` values map to a `static final` member variable, prefixed by an underscore (_), for example, `_apple = 0;` these underscored values can be used in switch statements and also to represent `enum`s as integers.
3. Each value in the `enum` object also maps to a `public static final` member variable with the same name as the value.
4. The `value()` method retrieves the integer value associated with each value of the `enum`. The integer values are assigned sequentially, beginning with 0.
5. The `from_int()` method returns the value `enum` object from a specified integer value.

A holder class is also generated for `enum`s, in this case `FruitHolder`.

Since only a single instance of an `enum` value object exists, the default `java.lang.Object` implementation of `equals()` and `hash()` can be used on objects associated with the `enum`.

Structs

A `struct` type allows you to form an aggregate structure of variables, which may be of the same or different types.

Consider the `struct` in the following IDL definition:

```
// IDL
interface Clock {
   struct Time {
      short hour;
      short minute;
      short second;
   };

   void updateTime (in Time current);
```

```
    void currentTime (out Time current);
    };
```

The rules by which an IDL `struct` is mapped to Java are illustrated in the Java mapping for the `Time` `struct`.

The IDL to Java compiler maps the `Time` structure as follows:

```
// Java generated by the OrbixWeb IDL compiler
// Time.java
package ClockPackage;
1    public final class Time {
2       public short hour;
        public short minute;
        public short second;

3       public Time () {}
4       public Time (short hour, short minute,
                     short second) {
           ...
        }
     }
```

1. The IDL `struct` called `Time` maps to a `final` Java class of the same name.
2. The `Time` class contains one instance variable for each field (`hour`, `minute`, `second`) in the structure.
3. There are two constructors (in this case, `Time`) for the structure class: the first, `Time()`, takes no arguments, and initializes all fields in the structure to null or zero.
4. The second constructor takes the fields in the structure as arguments `Time(short hour, short minute, short second)`, and initializes the structure.

The interface `Clock` maps to the Java Reference interface `Clock` as follows:

```
    // Java generated by the OrbixWeb IDL compiler
    // Clock.java
    import org.omg.CORBA.Object;
    import ClockPackage.Time;
1   import ClockPackage.TimeHolder;
```

```
    public interface Clock extends Object {
2       public void updateTime(Time current);
3       public void currentTime(TimeHolder current) ;
    }
```

1. Holder classes are generated for all `struct` types, with the name format `<type>Holder`, where `<type>` is the name of the `struct`, in this case `Time`.
2. The operations map to public Java methods of the same name, the `in` parameter mapping directly to the corresponding Java type, `Time`.
3. The `out` parameter is mapped to a `TimeHolder` type, to allow the values to be passed correctly.

Unions

IDL supports *discriminated* unions. A discriminated union consists of a discriminator and a value: the discriminator indicates what type the value holds.

Note: Union types do not exist in Java, and it is recommended that you only use the union mapping to support legacy IDL that already makes use of unions.

Consider the following example:

```
    //IDL for account
    //example of a discriminated Union

    interface account {};
    interface currentAccount : account {};
    interface depositAccount : account {};

1   union accountType switch (short)
    {
    case 1: currentAccount curAcc;
    case 2: depositAccount depacc;
    default: account genAcc;
    };
```

1. Here, in the union `accountType`, the `switch` discriminator indicates which case label value is being held.

The IDL discriminated union defined above maps to Java as follows:

```
// Java generated by the OrbixWeb IDL compiler

public final class accountType {
1       public accountType() {}
2       public short discriminator() {
          ...
        }
3       public currentAccount curAcc() {
          ...
        }
4       public void curAcc(currentAccount value) {
          ...
        }
5       public void curAcc (currentAccount value,
                            short discriminator){
          ...
        }
        public depositAccount depacc() {
          ...
        }
        public void depacc(depositAccount value) {
          ...
        }
        public void depacc (depositAccount value,
                            short discriminator) {
            ...
        }
        public account genAcc() {
          ...
        }
        public void genAcc(account value) {
          ...
        }
        public void genAcc(account value, short discriminator) {
          ...
        }
}
```

1. The union `accountType` maps to a public final class of the same name, with a corresponding default constructor, `accountType()`.

2. The value returned by the `discriminator()` method indicates which variable in the union currently stores a value. You should check the value returned by this method to determine which accessor method should be used.
3. For each variable in the union, there is a corresponding accessor method of the same name (`curAcc()`, `depAcc` and the default `genAcc`) which retrieves the value held in the variable. The accessor method used in the application code is determined by the value returned by the `discriminator()` method.
4. The modifier methods for each variable in the union are used to automatically set the value for the `discriminator()` method.
5. An additional modifier method is available to set the value of variables for use in situations where more than one case label is used. Only one case label is used in this example, so this method is not relevant here.

In rare cases, where a variable has more than one corresponding case label, the simple modifier method for that variable sets the discriminator to the value of the first case label. The secondary modifier method allows an explicit discriminator value to be passed, which may be necessary if a variable has more than one case label. When the value of a variable corresponds to the `default` case label, the modifier method sets the discriminant to a unique value, distinct from other case label values.

Note: If you pass a bad discriminator value, the secondary modifier throws an exception.

The following code shows how to assign a `depositAccount`:

```
      // Java

1     depositAccount dep;
2     accountType accType = new accountType();
      ...

3     accType.depAcc (dep, (short)2);

      // Java
      currentAccount cur;
      depositAccount dep;
      account acc;
```

```
...

switch (accType.discriminator ()) {
   case 1: cur = accType.curAcc ();
   break;
   case 2: dep = accType.depAcc ();
   break;
   default: acc = accType.genAcc ();
}
```

1. Create a new `depositAccount` object.
2. Create an instance of the union type.
3. Pass the value for `depositAccount` using the modifier method.
4. Invoke the `discriminator()` method to retrieve the active value in the union.

Mapping for Strings

IDL bounded and unbounded `string`s map to the Java type `java.lang.String`. As a Java `String` is fundamentally unbounded, OrbixWeb checks the range of `String` parameter values passed as bounded strings to IDL operations. If the actual string length is greater than the bound value, the `org.omg.CORBA.MARSHAL` exception is thrown.

The IDL type `wstring`, which can represent the full range of UNICODE characters, also maps to the Java type `String`. Range violations for the IDL `string` types raise `CORBA::DATA_CONVERSION` and `CORBA::MARSHAL` exceptions.

IDL `string` parameters defined as `inout` or `out` map to Java method parameters of type `org.omg.CORBA.StringHolder`. This `Holder` class contains a Java `String` value, which you can update during the operation invocation.

Consider the following IDL definition:

```
// IDL
interface Customer {
   void setCustomerName (in string name);
   void getCustomerName (out string name);
};
```

This maps to the following Java Reference interface:

```
// Java generated by the OrbixWeb IDL compiler
import org.omg.CORBA.Object;
import org.omg.CORBA.StringHolder;

public interface Customer extends Object {
1     public void setCustomerName(String name) ;
2     public void getCustomerName(StringHolder name) ;
};
```

1. IDL operations are mapped to Java methods of the same name.
2. IDL `out` parameters are mapped to `StringHolder` types to allow parameter passing.

The `StringHolder` class available in the `org.omg.CORBA` package is as follows:

```
// Java
package org.omg.CORBA;

public class StringHolder {
   public String value;

   public StringHolder () {}

   public StringHolder (String value) {
      this.value = value;
   }
}
```

The following code demonstrates how a client application could invoke the IDL operations defined in the `Customer` interface:

```
// Java
Customer cRef;
String inName = "Chris";
String outName;
StringHolder outNameHolder = new StringHolder();

// Here, cRef is set to reference a
// Customer (code omitted).

cRef.setCustomerName (inName);
cRef.getCustomerName (outNameHolder);
```

```
outName = outNameHolder.value;
```

The server programmer receives the `StringHolder` variable as a parameter to the implementation method and simply assigns the required string to the `value` field.

Mapping for Sequences

IDL bounded and unbounded sequences are mapped to Java arrays of the same name. In the case of bounded sequences OrbixWeb performs bounds checking on the mapped array during any operation invocations. This check ensures that the array length is less than the maximum length specified for the bounded sequence. A `CORBA::MARSHAL` exception is raised when the length of a bounded sequence is greater than the maximum length specified in the IDL definition.

Both holder and helper classes are generated for each of these sequence types.

The following IDL definition provides an example of declaring IDL sequences:

```
// IDL
module finance {
   interface account {
      attribute string Name;
      attribute float AccNumber;
   };

   struct limitedAccounts {
      string bankSortCode<10>;
      // Maximum length of sequence is 50.
      sequence<account,50> accounts;
   };

   struct unlimitedAccounts {
   string bankSortCode<10>;
   // No maximum length of sequence.
   sequence<account> accounts;
   };
};
```

Given the above example, the IDL compiler produces the following generated classes, one for the bounded sequence, and another for the unbounded sequence:

```
// Java generated by the OrbixWeb IDL compiler
// Bounded sequence
package Finance;

1   public final class limitedAccounts {
2      public String bankSortCode;
3      public account[] accounts;
4      public limitedAccounts() {}
5      public limitedAccounts (String bankSortCode,
                               account[] accounts) {
          ...
       }
       ...
    }
```

1. An IDL `struct` maps to a Java `public final class` of the same name, in this case, `limitedAccounts`.
2. The `string` type is mapped to a Java member variable of type `String`.
3. The bounded sequence `account` is mapped to a Java array of the same name.
4. The `struct` has two constructors; the first of which is a null constructor.
5. The second constructor initializes the public member variables, `bankSortCode` and the `account` array.

Unbounded sequences are mapped in the same way as bounded sequences. However, bounds checking is not done on the mapped array during operation invocations.

Mapping for Arrays

IDL arrays map directly to Java arrays. However, Java arrays are not bounded, therefore OrbixWeb explicitly checks the bound of an array when an operation is called with the array as an argument.

Arrays are fixed length objects so a `CORBA::MARSHAL` exception is thrown if the length of an array is not equal to the length specified in the IDL file. The length of the array can be made available in Java by bounding the array with an IDL constant, which is mapped according to the rules specified for constants.

A holder class for the array is also generated, with the format `<array name>Holder`.

As a simple example, consider the following IDL definition for an array:

```
// IDL
typedef short BankCode[3];

interface Branch {
   attribute string location;
   attribute BankCode code;
};
```

This maps to:

```
// Java generated by the OrbixWeb IDL compiler
// in file Branch.java
import org.omg.CORBA.Object;

public interface Branch extends Object {
    public String location();
    public void location(String value);
    public short[] code();
    public void code(short[] value);
}
```

Mapping for Constants

The way IDL constants map to Java depends on whether or not they are declared within an interface.

Constants Defined within an IDL Interface

An IDL constant defined *within* an interface maps to a `public static final` member of the corresponding Java Reference interface generated by the IDL to Java compiler.

For example, consider the following IDL:

```
// IDL
interface ConstDefIntf {
   const short MaxLen = 4;
};
```

This maps to the following Java class:

```
// Java generated by the OrbixWeb IDL compiler
// in file ConstDefInt.java
import org.omg.CORBA.Object;

public interface ConstDefIntf extends Object {
    public static final short MaxLen = 4;
}
```

You can then access the constant by scoping with the Java class name, for example:

```
// Java
short len = ConstDefIntf.MaxLen;
```

Constants Declared outside an IDL Interface

Those constants which are declared *outside* an IDL interface are mapped to a `public interface` with the same name as the constant and containing a `public static final` field, named `value`. The `value` field holds the value of the constant. Since these Java classes are only required at compile time, the Java compiler normally inlines the value when the classes are used in other Java code.

Consider the following IDL:

```
// IDL
module ExampleModule {
   const short MaxLen = 4;
};
```

This maps to the following Java class:

```
// Java generated by the OrbixWeb IDL compiler
package ExampleModule;

public interface MaxLen {
    public static final short value = 4;
}
```

You can then access the constant by scoping with the Java interface name, for example:

```
// Java
short len = ExampleModule.MaxLen.value;
```

Mapping for Typedefs

Java has no language construct equivalent to the IDL `typedef` statement. The Java mapping resolves the `typedef` to the corresponding base IDL type, and maps this base type according to the IDL Java Mapping. A `Helper` class for the declared type is also produced. If the type is a sequence or array, `Holder` classes are also generated for the declared types.

All distinct IDL types, including those declared as `typedefs`, require a unique Repository ID within the Interface Repository. For this reason, `Helper` classes for the types declared as `typedefs` are automatically generated with the format:

`<declared Type>Helper`

For example, consider the following `typedef` declaration:

```
// IDL
struct CustomerDetails {
   string Name;
   string Address;
};
typedef CustomerDetails BankCustomer;
```

The `CustomerDetails` structure maps to a Java class as described in "Mapping for Constants" on page 127. The `typedef` statement results in an additional `BankCustomerHelper` class.

Mapping for Exception Types

CORBA defines two categories of exception type:

- IDL standard system exceptions.
- IDL user-defined exceptions.

System Exceptions

IDL standard system exceptions are mapped to `final` Java classes that extend `org.omg.CORBA.SystemException`. These classes provide access to the IDL major and minor exception code, as well as a string describing the reason for the exception. IDL system exceptions are *unchecked* exceptions. This is because the class

`org.omg.CORBA.SystemException` is derived from `java.lang.Runtime.Exception`.

For further information on the mapping of IDL System Exceptions to Java refer to the *OrbixWeb Programmer's Reference*.

User-Defined Exceptions

An IDL user-defined exception type maps to a `final` Java class that derives from `org.omg.CORBA.UserException`, which in turn derives from `java.lang.Exception.` `Helper` and `Holder` classes are also generated. IDL user-defined exceptions are *checked* exceptions.

If the exception is defined within an IDL interface, its Java class name is defined within the interface package, called `<interface name>Package`. Where a module has been defined, the Java class name is defined within the scope of the Java package corresponding to the IDL module enclosing the exception.

Consider the following IDL user-defined exception:

```
//IDL
module Exceptions {
   interface Illegal {
      exception reject {
         string reason;
         short s;
      };
   };
};
```

The `reject` exception maps as follows:

```
// Java generated by the OrbixWeb IDL compiler
// in file reject.java
import org.omg.CORBA.UserException;

public final class reject extends UserException {
    public String reason;
    public short s;
    public reject() {
        ...
    }
```

```
    public reject(String reason, short s) {
        ...
    }
    ...
}
```

The mapping of the `reject` exception illustrates the rules used by the IDL to Java compiler when mapping exception types. The `reject` exception maps to the `final` class `reject`, which extends `org.omg.CORBA.UserException`. Instance variables for the fields `reason` and `s`, defined in the exception, are also provided. There are two constructors in the mapped exception: `reject()` is the default constructor and the `reject(String reason, short s)` constructor initializes each exception member to the given value.

Now consider an interface with an operation that can raise a `reject` IDL exception:

```
// IDL
interface bank {
   exception reject {
      ...
   };

   account newAccount() raises (reject);
};
```

A server can throw a `bankPackage.reject` exception in exactly the same way as a standard Java exception.

An OrbixWeb client can test for such an exception when invoking the `newAccount()` operation as follows:

```
// Java
bank b;
account a;

...

try {
   a = b.newAccount ();
}
catch (bankPackage.reject rejectEx) {
   system.out.println ("newAccount() failed");
   system.out.println ("reason for failure = " +
      rejectEx.reason);
```

```
    ...
}
```

OrbixWeb exception handling is described in detail in "Exception Handling" on page 295.

Naming Conventions

IDL identifiers are mapped to an identifier of the same name in Java. There are, however, certain names which are reserved by the Java mapping. When these occur within IDL definitions, the mapping uses a prefixed underscore ('_') to distinguish the mapped identifier from a reserved name.

Reserved names in Java include the following:

- Java keywords.

  If an IDL definition contains an identifier that exactly matches a Java keyword, the identifier is mapped to the name of the identifier preceded by '_' as follows:

  `_<keyword>`

  Refer the Java Language Specification for more details about Java keywords.

- The Java class `<type>Helper`, where `<type>` is the name of an IDL user-defined type.
- The Java class `<type>Holder`, where `<type>` is the name of an IDL user-defined type.

  When a `typedef` alias is used, the resulting Java class has the format `<alias>Holder`.

- The Java classes `<basicJavaType>Holder`, where `<basicJavaType>` represents a Java basic type to which an IDL basic type is mapped.

  Refer to Table 1 on page 92 for details of these types.

- The Java package name `<interface>Package`, where `<interface>` is the name of an already-defined IDL interface.

Parameter Passing Modes and Return Types

Table 2 shows the mapping for the IDL parameter passing modes and return types. Refer to "Holder Classes and Parameter Passing" on page 100 for more details. All non-user defined type Holders are in `org.omg.CORBA`.

IDL Type	In	Inout	Out	Return
Basic Types				
short	short	ShortHolder	ShortHolder	short
long	int	IntHolder	IntHolder	int
unsigned short	short	ShortHolder	ShortHolder	short
unsigned long	int	IntHolder	IntHolder	int
long long	long	LongHolder	LongHolder	long
unsigned long long	long	LongHolder	LongHolder	long
float	float	FloatHolder	FloatHolder	float
double	double	DoubleHolder	DoubleHolder	double
boolean	boolean	BooleanHolder	BooleanHolder	boolean
char	char	CharHolder	CharHolder	char
wchar	char	WcharHolder	WcharHolder	char
octet	byte	ByteHolder	ByteHolder	byte
any	Any	AnyHolder	AnyHolder	Any
IDL User-Defined Types				
enum	<type>	<type>Holder	<type>Holder	<type>
struct	<type>	<type>Holder	<type>Holder	<type>
union	<type>	<type>Holder	<type>Holder	<type>

Table 2: *Mapping for Parameters and Return Values*

IDL Type	In	Inout	Out	Return
string	String	StringHolder	StringHolder	String
wstring	String	WstringHolder	WstringHolder	String
sequence	array	<type>Holder	<type>Holder	array
array	array	<type>Holder	<type>Holder	array
Pseudo-IDL Types				
NamedValue	NamedValue	NamedValueHolder	NamedValueHolder	NamedValue
TypeCode	TypeCode	TypeCodeHolder	TypeCodeHolder	TypeCode
object reference	<type>	<type>Holder	<type>Holder	<type>

Table 2: *Mapping for Parameters and Return Values*

7

Using and Implementing IDL Interfaces

This chapter describes how clients access objects through IDL interfaces and how servers create objects that implement those interfaces. A detailed banking example illustrates how to use and implement CORBA objects.

Overview of an Example Application

The example described in this chapter is a banking application. An OrbixWeb server creates a single distributed object which represents a bank. This object manages other distributed objects which represent customer accounts at the bank.

A client contacts the server by getting a reference to the bank object. The client then calls operations on the bank object that instruct the bank to create new accounts for specified customers. The bank object creates account objects in response to these requests and returns them to the client. The client can then call operations on these new account objects.

This application design, in which one type of distributed object acts as a factory for creating another type of distributed object, is very common in CORBA.

The sample code described in this chapter is available in the `demos/bank_demo` directory of your OrbixWeb installation.

Overview of the Programming Steps

The programming steps are outlined as follows:

1. Define the IDL interfaces to objects used by the application.
2. Compile the IDL using the IDL to Java compiler.
3. Implement the IDL interfaces bank and account.
4. Write a server application that creates bank and account objects.
5. Write a client application that accesses bank and account objects.
6. Run an OrbixWeb daemon process.
7. Register the server in the Implementation Repository.
8. Run the client.

Subsequent chapters add further functionality to the bank and account interfaces defined in this chapter. At this stage, the basic interfaces are sufficient to illustrate the main points. Examples shown in later chapters allow operations to raise user-defined exceptions.

Defining IDL Interfaces to Application Objects

The example which follows implements a simple banking application. This example creates a server which administers and manages bank account objects. The functionality required is defined by the following IDL interface definitions:

```
// IDL
// In file "bank_demo.idl".

// A simple bank account.
interface account {
   readonly attribute float balance;

   void makeLodgement(in float f);
   void makeWithdrawal(in float f);
};
...
```

```
// bank manufactures bank accounts.
interface bank {
   // Create a new account for the given name.
   account newAccount (in string name);

   // Delete an account.
   void deleteAccount(in account a);
};
```

On the server-side, the example creates a server which implements a single `bank` object. This accepts operation invocations such as `newAccount()` from clients. In this example, all objects are located in a single server. In an actual system several servers may be used. A server can manage the objects of different interfaces.

Compiling IDL Interfaces

It is assumed that the `bank_demo.idl` source file is compiled using the following IDL compiler command:

```
idl -jP bank_demo bank_demo.idl
```

Refer to Chapter 6, "IDL to Java Mapping" on page 91 for more details on the classes generated by the IDL to Java compiler.

Implementing the Interfaces

OrbixWeb supports two mechanisms for relating an implementation class to its IDL interface:

- *The ImplBase approach*
- *The TIE approach*

The TIE approach is preferred for the majority of implementations in Java. This is due to the restriction of single inheritance of classes in Java which limits the ImplBase approach. Refer to "Comparison of the ImplBase and TIE Approaches" on page 165 for more details. Both approaches can, however, be used in the same server, if required.

This section briefly describes how an interface may be implemented using both of these approaches. It then steps through an example implementation using the TIE approach. For an example of the ImplBase approach refer to "Implementing the IDL Interface" on page 16.

Note: The choice of implementation method in an OrbixWeb server does not affect the coding of client applications.

The TIE Approach

The TIE approach to defining an implementation class is shown in Figure 15 on page 139.

Using the TIE approach, you can implement the IDL operations and attributes in a class which does *not* inherit from the automatically generated ImplBase class. Instead, use the automatically generated Java TIE class to *tie together* the implementation class and the IDL interface.

The IDL compiler generates a Java TIE class for each IDL interface. The name of the Java TIE class takes the form of `_tie_` appended to the name of the interface. For example, the IDL compiler generates the TIE class `_tie_account` for the IDL interface type `account`. An object which implements the IDL interface is passed as a parameter to the constructor for the TIE class.

To use the TIE approach you should define a new class, `accountImplementation`, which implements the operations and attributes defined in the IDL interface. This class need not inherit from any automatically generated class, however, it must implement the Java interface `_accountOperations`.

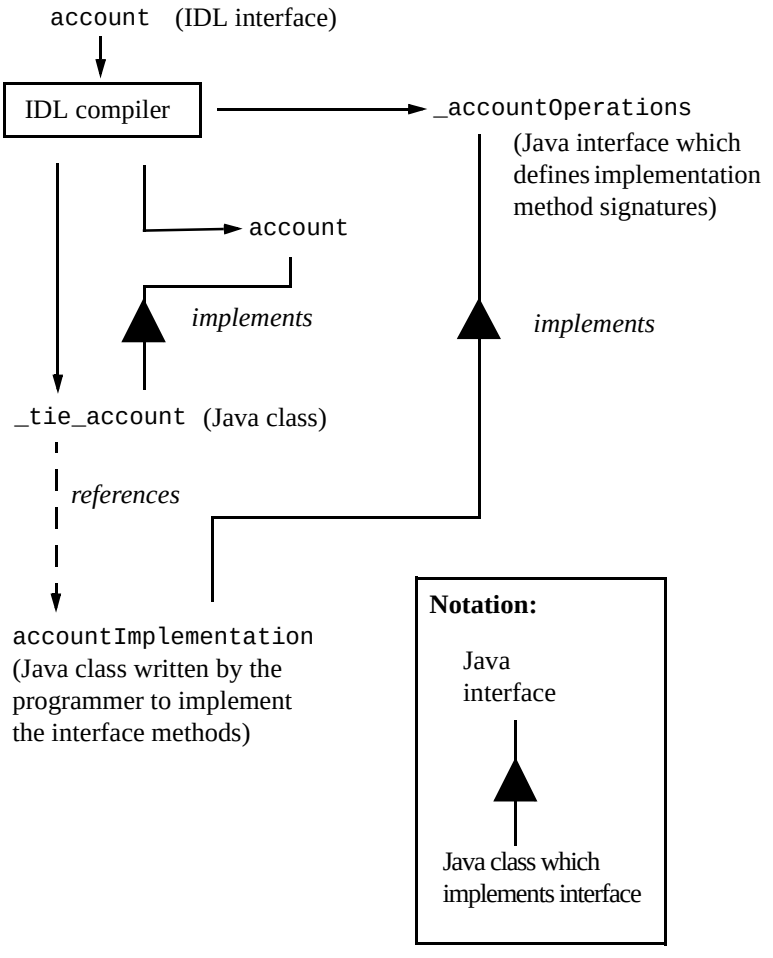

Figure 15: *The TIE Approach to Defining an Implementation Class*

Then instantiate an object of type `_tie_account`, passing an object of type `accountImplementation` to the constructor. A TIE object is thus created which delegates incoming operation invocations to the methods of your `accountImplementation` object.

Interface _accountOperations is generated by the IDL compiler as follows:

```
// Java generated by the OrbixWeb IDL compiler

package bank_demo;
public interface _accountOperations  {
   public float balance();

   public void makeLodgement (float f) ;
   public void makeWithdrawal (float f) ;
}
```

The ImplBase Approach

For each IDL interface, OrbixWeb also generates an abstract Java class, named _<type>ImplBase, where <type> represents the name of a user-defined IDL interface. For example, the class _accountImplBase is generated for the IDL interface account. To indicate that a Java class implements a given IDL interface, that class should inherit from the corresponding ImplBase class. This approach is termed the *ImplBase Approach*, and is the implementation method defined by the CORBA specification.

Because each ImplBase class is the Java equivalent of an IDL interface, a class that inherits from this implements the operations of the corresponding IDL interface.

To support the use of the ImplBase approach, the OrbixWeb IDL compiler produces the Java interface account and the Java class _accountImplBase.

The ImplBase approach is used in Chapter 2 "Getting Started with Java Applications" on page 13 for the implementation of the Grid interface. In this chapter and in the rest of this guide, the TIE approach is used to implement IDL interfaces. The TIE approach is the method of choice for the majority of Java applications.

Developing the Server Application

In this section, a simple implementation of the banking application is used to illustrate the TIE approach. The error-handling necessary for a full banking application has been omitted; for example, checking if the account is overdrawn. Refer to Chapter 15,"Exception Handling" on page 295 for more details.

The following implementation classes are coded:

`bankImplementation`	Implements the `bank` interface.
`accountImplementation`	Implements the `account` interface.

With the TIE approach, an implementation class does not have to inherit from any particular base class. Instead, the implementation class must implement the Java `Operations` interface generated by the IDL compiler from the IDL interface definition. You should notify OrbixWeb that this class implements the IDL interface by creating an object of the TIE class, which is also generated by the IDL compiler.

Recall the IDL definitions for interfaces `account` and `bank`:

```
// IDL
// In file "bank_demo.idl".

// A simple bank account.
interface account {
   readonly attribute float balance;

   void makeLodgement(in float f);
   void makeWithdrawal(in float f);
};
...
// Manufacures bank accounts.
interface bank {
   // Create a new account for the given name.
   account newAccount (in string name);
   ...
   // Delete an account.
   void deleteAccount(in account a);
};
```

Account Class Implementation

The account class implementation implements the Java `Operations` interface generated by the IDL Compiler. This class is coded as follows:

```
// Java
// In file accountImplementation.java.

package bank_demo;

import org.omg.CORBA.SystemException;
import org.omg.CORBA.ORB;
import IE.Iona.OrbixWeb._CORBA;
import IE.Iona.OrbixWeb._OrbixWeb;
...

// The account implementation class.
class accountImplementation
   implements _accountOperations {

   public accountImplementation () {
      m_balance = 0;
      m_name = "";
   }

   public accountImplementation
      (float initialBalance, String name) {
      m_balance = initialBalance;
      m_name = name;
   }

   public String get_name () {
      return m_name;
   }
```

```
   public float balance () {
      return m_balance;
   }

   public void makeLodgement (float f) {
      m_balance += f;
   }

   public void makeWithdrawal(float f) {
      m_balance -= f;
   }

   String m_name;
   float m_balance;
}
```

You can then instantiate a _tie_account object as follows:

```
// Java
package bank_demo;
   ...

account accImpl = null;

try {
   accImpl = new _tie_account
      (new accountImplementation ());
}
catch (SystemException se) {
   ...
}
```

Bank Class Implementation

Using the TIE approach, you can write the code for the bank implementation class as follows:

```
// Java
// In file bankImplementation.java.
// The bank creates accounts and maintains
// a Vector of all accounts created.

package bank_demo;

import org.omg.CORBA.SystemException;
import org.omg.CORBA.ORB;
import IE.Iona.OrbixWeb._CORBA;
import IE.Iona.OrbixWeb._OrbixWeb;
import java.util.Vector;
...

class bankImplementation
   implements _bankOperations {
   IE.Iona.OrbixWeb.CORBA.ORB orb = null;

      public bankImplementation (org.omg.CORBA.ORB orb)
         throws SystemException  {
         super();
         this.orb = _OrbixWeb.ORB(orb);
      }

// Internal record() operation used to add new
// accounts to the Vector
      void record
         (String name,accountImplementation p) {
            AccList.addElement(p);
            nameList.addElement(name);
      }

// newAccount() creates a new account and adds it
// to the account Vector.
      public account newAccount(String name) {
      ...
         System.out.println
            ("Creating Account for " + name);
```

```
            accountImplementation accImpl = null;
         ...

         try {
            accImpl = new accountImplementation(0,name);
         }
         catch (SystemException se) {
            System.out.println("Exception : " +
               se.toString());
         }
         account acc = new _tie_account(accImpl);
         record(name , accImpl);
         return acc;
      }
         ...

      public void deleteAccount(account a) {
         AccList.removeElement((accountImplementation)
            (a._deref()));
         nameList.removeElement((accountImplementation)
            (a._deref())).get_name());
      }

      // account object list
      Vector AccList = new Vector();
      //name list
      Vector nameList = new Vector();
   }
```

The method newAccount( ) returns an object that implements Java interface account. This IDL generated type defines the client view of the IDL interface account.

main() Method and Object Creation

This section shows a sample `main()` method of a banking server application, using the TIE approach.

The code instantiates three `bank` objects with associated TIE objects. Two of the `bank` objects are assigned markers, and client applications may request either of these when obtaining a reference by specifying the relevant marker in the `bind()` call. Refer to Chapter 8, "Making Objects Available in OrbixWeb" on page 171 for more details on markers.

```
// Java
// In file javaserver1.java

package bank_demo;

import IE.Iona.OrbixWeb._CORBA;
import IE.Iona.OrbixWeb._OrbixWeb;
import org.omg.CORBA.SystemException;
import java.util.Vector;

// The bank_demo server.
public class javaserver1 {
   public static void main (String args[]) {

      // Initialize the ORB and implicitly
      // call ORB.connect().
      org.omg.CORBA.ORB orb = null;
      orb = org.omg.CORBA.ORB.init (args,null);

      bank AIBbankImpl= null;
      bank serverImpl = null;
      bank CGbankImpl = null;

      // create 3 bank objects
      try {
         serverImpl = new _tie_bank
                        (new bankImplementation (orb));
         AIBbankImpl = new _tie_bank
                        (new bankImplementation
                                       (orb),"AIB");
```

```
            CGbankImpl  = new _tie_bank
                            (new bankImplementation
                               (orb),"College Green AIB");

            // Pause the server to prevent exiting
            // and allow incoming invocations.
            try {
               Thread.sleep (50000);
            }
            catch (InterruptedException ex) {}

            // Stop all event processing.
            orb.disconnect (CGbankImpl);
            orb.disconnect (AIBbankImpl);
            orb.disconnect (serverImpl);
         }
         catch (SystemException se) {
            System.out.println
               ("Exception in new bankImplementation: ");
            System.out.println (se.toString());
            System.exit (1);
         }
          _OrbixWeb.ORB(orb).shutdown(true);
      }
   }
```

Object Initialization and Connection

An implementation object must be connected to the OrbixWeb runtime before it can handle incoming operation invocations.

There are two means of connecting implementation objects to the OrbixWeb runtime. These are as follows:

- Using `ORB.connect()` and `ORB.disconnect()`

  These methods are the CORBA-defined way of connecting an implementation to the runtime.

- Using `_CORBA.Orbix.impl_is_ready`

  This is an OrbixWeb-specific way of connecting implementation objects to the runtime.

ORB.connect() and ORB.disconnect()

The OMG standard way of connecting an implementation to the runtime is to use `org.omg.CORBA.ORB.connect()`. The OrbixWeb runtime can continue to make invocations on the implementation until it is disconnected using `org.omg.CORBA.ORB.disconnect()`. Refer to the API Reference on interface BOA in the *OrbixWeb Programmer's Reference* for more details.

As an example, consider the following code, which instantiates a `bank` implementation object and connects it to the runtime. The implementation object is disconnected at a later stage.

```
import org.omg.CORBA.ORB;

   ORB orb = ORB.init(args,null);

   bank mybank =
        new _tie_bank(new bankImplementation());

   orb.connect(bank);
   ...
   orb.disconnect(bank);
```

Note: `ORB.connect()` is automatically called when you instantiate an OrbixWeb object. However, for strict CORBA compliance, you should explicitly call `ORB.connect()` in your application code.

_CORBA.Orbix.impl_is_ready

A server is normally coded so that it initializes itself and creates an initial set of objects. It then calls `_CORBA.Orbix.impl_is_ready()` to indicate that it has completed its initialization and is ready to receive operation requests on its objects. `_CORBA.Orbix` is a static object (of interface BOA or class ORB) which is used to communicate directly with OrbixWeb to determine or change its settings.

The `impl_is_ready()` method normally does not return immediately. It blocks the server until an event occurs, handles the event, and then re-blocks the server to wait for another event.

The method impl_is_ready() consists of four overloaded methods, as follows:

```
// Java
// In package IE.Iona.OrbixWeb.CORBA
// in interface BOA.

public void impl_is_ready ();

public void impl_is_ready (String serverName);

public void impl_is_ready (int timeout);

public void impl_is_ready
   (String serverName, int timeout);
```

serverName

The serverName parameter to impl_is_ready() is the name of a server as registered in the Implementation Repository.

When a server is launched by the OrbixWeb daemon process, the server name is already known to OrbixWeb and therefore does not need to be passed to impl_is_ready(). However, when a server is launched manually, the server name must be communicated to OrbixWeb. The normal way to do this is as the first parameter to impl_is_ready(). To allow a server to be launched either automatically or manually, it is recommended, therefore, that you specify the serverName parameter.

By default, servers must be registered in the Implementation Repository, using the putit command. Therefore, if an unknown server name is passed to impl_is_ready(), the call is rejected. However, the OrbixWeb daemon can be configured to allow unregistered servers to be run manually. Refer to Chapter 12, "Registration and Activation of Servers" on page 251 for more details on the OrbixWeb daemon and the putit command.

timeout

The impl_is_ready() method returns only when a timeout occurs or an exception occurs while processing an event. The timeout parameter indicates the number of milliseconds to wait between events. A timeout occurs if OrbixWeb has to wait longer than the specified timeout for the next event. A timeout of zero causes impl_is_ready() to process an event, if one is immediately available, and then return.

A server can time out either because it has no clients for the timeout duration, or because none of its clients uses it for that period. The system can also be instructed to make the timeout active only when the server has no current clients. The server should remain running as long as there are current clients. This is supported by the method `setNoHangup()`, defined in interface `BOA`. Refer to the *OrbixWeb Programmer's Reference* for more details on interface `BOA`.

You can explicitly pass the default timeout as `_CORBA.IT_DEFAULT_TIMEOUT`. The default value of the `_CORBA.IT_DEFAULT_TIMEOUT` parameter is one minute. You can specify an infinite timeout by passing `_CORBA.IT_INFINITE_TIMEOUT = -1`.

Comparison of Methods for Connecting to the ORB

This section outlines some of the merits and drawbacks of the `impl_is_ready()` and `ORB.connect()` / `ORB.disconnect()` methods for connecting to the ORB.

The primary advantage of using `_CORBA.Orbix.impl_is_ready()` is that it allows server registration and event processing to be decoupled. This gives the programmer who implements the server more control over event processing. This is the BOA approach familiar to users of previous versions of OrbixWeb.

The `ORB.connect()` / `ORB.disconnect()` approach complies with the CORBA specification defined in the OMG IDL to Java mapping. Using this approach, OrbixWeb implicitly connects an implementation object to the runtime when the object is instantiated. By default, when `ORB.connect()` is first called in a server, a background thread that processes events is created, and the server makes itself known to the OrbixWeb daemon.

Correspondingly, calling `ORB.disconnect()` on the last registered object stops all event processing. You can disable this behaviour by setting the configurable item `IT_IMPL_READY_IF_CONNECTED` to `false`.

When this approach is used in servers launched persistently, the server has no means of specifying a server name. The server name must be specified using `_CORBA.Orbix.setServerName()` or by passing it on the command line to the Java VM using `-DOrbixWeb.server_name`.

By default, even if the target object has been disconnected, the server continues processing requests until the last object has been disconnected. This can result, for example, in an `INV_OBJREF` exception to a client in response to an incoming request for a disconnected object. It is important, therefore, to explicitly disconnect all objects when you want your server to exit. It is also important to disconnect all objects so that they can call their loaders,

if any exist, in order to save themselves. Refer to Chapter 24, "Loaders" on page 453 for more details.

In the case of *out-of-process* servers, where each launched server has its own system process, you can disconnect all objects using the following call:

```
_OrbixWeb.ORB(orb).shutdown(true);
```

In the case of *in-process* servers, this method has no effect. Refer to Chapter 7, "The OrbixWeb Java Daemon" on page 271 for details on in-process servers. By default, servers are activated out-of-process.

You can combine the two approaches to connecting to the ORB. In fact, if you call BOA event processing operations, a combined approach is used. `ORB.connect()` is implicitly called on when the implementation object is instantiated. Also, in OrbixWeb, several threads can concurrently call `processEvents()`.

Note: Disconnecting the last object by default causes all BOA event processing calls to exit.

Construction and Markers

The name of an OrbixWeb object includes its server name, interface, and a unique name within that server and interface.

In the bank application, if a client needs to find an individual `bank` object of a given name, you can assign a meaningful marker name to each `bank` object, and then allow clients to specify one of these marker names when calling the `bankHelper.bind()` method. "Binding to Objects in OrbixWeb Servers" on page 204 shows how to do this.

The best way for a client to obtain an object reference for a particular `account` object is for the `bank` to provide an IDL operation that takes the `account` holder name and returns an `account` object reference.

Developing the Client Application

From the point of view of the client, the functionality provided by the banking service is defined by the IDL interface definitions. A typical client program locates a remote object, obtains a reference to the object, and then invokes operations on the object.

These three concepts (object location, obtaining a reference, and remote invocations) are important concepts in distributed systems. This section discusses developing the client application in terms of these concepts.

- *Object location* involves searching for an object among the available servers on available nodes. The CORBA defined way to do this is to use the Naming Service.
- *Obtaining a reference* involves establishing the facilities required to make remote invocations possible. This involves setting up a proxy. A reference to the proxy can then be returned to the client. Obtaining a reference is also termed *binding* to an object.
- *Remote invocations* in OrbixWeb occur when normal Java method calls are made on proxies.

Refer to Chapter 8 "Making Objects Available in OrbixWeb" on page 171 for more information on object references and object location.

These three concepts are illustrated in the following code extract from a client application which uses the banking service:

```
// Client application code
// In file javaclient1.java

package bank_demo;

import org.omg.CORBA.SystemException;
import org.omg.CORBA.ORB;
...

public class javaclient1 {
   public static void main(String args[]) {

      org.omg.CORBA.ORB orb = ORB.init (args,null);

      // create variables to hold proxy objects
      bank mybank = null;
```

```
        account currAccount = null;
        String hostname = null;
        ...

        // Search for an object offering the bank
        // server and construct a proxy.
        try {
           System.out.println
              ("Attempting to bind to :bank on "+hostname);
           mybank = bankHelper.bind (":bank",hostname);
        }
        catch (org.omg.CORBA.SystemException ex) {
           System.out.println
              ("Exception during bind : " + ex.toString());
         }
        System.out.println
           ("Connection to " + hostname + " succeeded.\n");

        try {
           currAccount = mybank.newAccount ("chris");
           System.out.println ("Account created ");
           currAccount.makeLodgement ((float)56.90);
           System.out.println
              ("Balance of first account is ");
           System.out.println (""+currAccount.balance());
           mybank.deleteAccount (currAccount);
        }
        catch (SystemException ex) {
           System.out.println
              ("Unexpected system exception :
                 "+ ex.toString());
           System.exit (1);
        }
        ...
     }
     ...
}
```

Object Location

The static method `bankHelper.bind()` requests OrbixWeb to search for an object offering the `bank` IDL interface. The IDL compiler generates six overloaded `bind()` methods for each IDL interface. In the case of the `bank` interface, these methods are defined as follows:

```
// Use Helper class code
// In file bankHelper.java
// Java generated by the OrbixWeb IDL compiler

package bank_demo;

import IE.Iona.OrbixWeb._OrbixWeb;
...

public class bankHelper {
     ...
     public static final bank bind() {
             ...
     }

     public static final bind
        (org.omg.CORBA.ORB orb) {
           ...
     }

     public static final bank bind
        (String markerServer) {
           ...
     }

     public static final bank bind
        (String markerServer, org.omg.CORBA.ORB orb) {
           ...
     }

     public static final bank bind
        (String markerServer, String host) {
           ...
     }
```

```
        public static final bank bind
           (String markerServer, String host,
                        org.omg.CORBA.ORB orb) {
                 ...
        }

        public static bank narrow(Object _obj) {
           ...
        }
     }
  }
```

The parameters to `bind()` are described in detail in the section "Tabular Summary of bind() Parameters" on page 209.

In the bank example, the client specifies a host at which to contact the server. If a host is not specified, OrbixWeb makes an implicit call to the *default locator* to find a host where the required server has been registered. Chapter 25 "Locating Servers at Runtime" on page 473 describes the functionality of the OrbixWeb locator mechanism.

The OrbixWeb `bind()` methods provide two alternatives to the default locator when attempting to locate a host on which a given server resides:

- Locate the server host in advance of a call to `bind()`, and then specify the known host name to the `bind()` method, as described in "Binding to Objects in OrbixWeb Servers" on page 204.
- Override the default locator with a user-defined locator implementation, as described in Chapter 25 "Locating Servers at Runtime" on page 473.

The parameter `markerServer` allows the object marker and server name to be specified in the `bind()` call. In the example, no object marker is specified, so OrbixWeb can select any available object that matches the remaining location parameters. The `markerServer` value "`:bank`" instructs OrbixWeb to search for the required object in the `bank` server.

Binding

OrbixWeb supports collocation of clients and servers in a single address space. However, in the bank example the client and server applications are distributed. Consequently, the call to `bind()` in the client constructs a proxy for the specified object. The `bind()` method returns a reference to the proxy object of type `bank`. The Java methods of this reference type define the client view of the `bank` IDL interface.

There are two case in which this `bind()` call does not create a proxy for the specified object:

- If a proxy for the object already exists in the client address space, this existing proxy is returned.
- If a system exception is thrown during the `bind()` call, the return value is undefined.

Note: Calling `bind()` is not always required before communicating with a particular object. If a call to another object returns a reference to a remote object, this results in the creation of a proxy for this remote object. For example, operation `newAccount()` returns a reference to an `account` object.

The OrbixWeb mechanism of binding client references to server objects is discussed in detail in "Binding to Objects in OrbixWeb Servers" on page 204.

Remote Invocations

The proxy object reference returned by `bind()` provides access to remote `bank` operations using the Java methods defined on interface `bank`. The client can invoke these operations by calling the equivalent Java methods on the proxy object. The proxy is responsible for forwarding the invocation requests to the target server implementation object and returning results to the client.

The Java interfaces `account` and `bank` are generated by the IDL compiler. These interfaces define the Java client view of the IDL `account` and `bank` interfaces.

The code for interface account is as follows:

```
// Java generated by the IDL compiler

package bank_demo;

public interface account
   extends org.omg.CORBA.Object {
      public float balance();

      public void makeLodgement(float sum) ;
      public void makeWithdrawal(float sum) ;
}
```

The code for interface bank is as follows:

```
// Java generated by the IDL compiler

package bank_demo;

public interface bank
      extends org.omg.CORBA.Object {
         public bank_demo.account newAccount
                                 (String name) ;
         public void deleteAccount
                         (bank_demo.account a) ;
}
```

All three Java types inherit from the Java interface `org.omg.CORBA.Object`. This is an OrbixWeb interface which defines functionality common to all IDL object reference types. Refer to the API Reference in the *OrbixWeb Programmer's Reference* on `org.omg.CORBA.Object` for further information on this extra functionality.

Registration and Activation

The last step in developing and installing the bank application is to register the `bank` server.

Running the OrbixWeb Daemon

Before registering the server you should ensure that an OrbixWeb daemon process (`orbixd` or `orbixdj`) is running on the server machine.

To run the OrbixWeb Java Daemon, type the `orbixdj` command from the `bin` directory of your OrbixWeb installation.

To run the OrbixWeb Daemon, type the `orbixd` command from the `bin` directory of your OrbixWeb installation.

In Windows, you can also start a daemon process by clicking on the appropriate menu item from the OrbixWeb menu.

The Implementation Repository

The OrbixWeb Implementation Repository records the server name and the details of the Java class which should be interpreted in order to launch the server. Implementation Repository entries consist of the class name, the class path, and any command line arguments which the class expects.

Every node in a network which runs servers must have access to an Implementation Repository. Implementation repositories can be shared using a network file system.

You can register a server in the Implementation Repository using the `putit` command, which takes the following simplified form:

```
putit [putit switches] -java serverName
      -classpath <class path> <class name>
      [command-line-args-for-server]
```

For example, you could register the `bank` server as follows:

```
putit -java bank bank_demo.javaserver1
```

The class `bank_demo.javaserver1` is then registered as the implementation code of the server called `bank` at the current host. The `putit` command does not cause the specified server class to be interpreted. The Java interpreter can be explicitly invoked on the class, or the OrbixWeb daemon causes the class to be interpreted in response to an incoming operation invocation. It uses the OrbixWeb `IT_DEFAULT_CLASSPATH` as its `CLASSPATH`

when searching for the class. You can specify an alternative CLASSPATH using the putit utility. Further information on the putit command is given in "Registration and Activation of Servers" on page 251.

Execution Trace

This section examines the events which occur when the bank server and client are run. The TIE approach is used to show the initial trace. This is followed by a comparison between the TIE approach and the ImplBase approach.

Server-Side

First, a server with name bank is registered in the Implementation Repository.

When an invocation arrives from a client, the OrbixWeb daemon launches the server by invoking the Java interpreter on the specified class. The server application creates a new TIE object, of type _tie_bank, for an object of class bankImplementation:

```
// Java
// In file javaserver1.java

package bank_demo;

import IE.Iona.OrbixWeb._CORBA;
import IE.Iona.OrbixWeb._OrbixWeb;
import org.omg.CORBA.SystemException;
import java.util.Vector;

// The bank_demo server.
public class javaserver1 {
   public static void main (String args[]) {

      // Initialize the ORB and implicitly
      // call ORB.connect().
      org.omg.CORBA.ORB orb = null;
      orb = org.omg.CORBA.ORB.init (args,null);
      bank AIBbankImpl= null;
      bank serverImpl = null;
      bank CGbankImpl = null;
```

```
        // create 3 bank objects
        try {
           serverImpl = new _tie_bank
                        (new bankImplementation
                                          (orb));
           AIBbankImpl = new _tie_bank
                           (new bankImplementation
                                        (orb),"AIB");
           CGbankImpl  = new _tie_bank
                           (new bankImplementation
                              (orb),"College Green AIB");

           // Pause the server to prevent exiting
           // and allow incoming invocations.
           try {
              Thread.sleep (50000);
           }
           catch (InterruptedException ex) {}

           orb.disconnect (CGbankImpl);
           orb.disconnect (AIBbankImpl);
           orb.disconnect (serverImpl);
        }
        catch (SystemException se) {
           System.out.println
              ("Exception in new bankImplementation: ");
           System.out.println (se.toString());
           System.exit (1);
        }
         _OrbixWeb.ORB(orb).shutdown(true);
     }
  }
```

Client-Side

The client first binds to any `bank` object, using `bind()`, for example:

```
// Java
// In file javaclient1.java.

package bank_demo;
...
public class javaclient1 {
   public static void main (String args[]) {
      org.omg.CORBA.ORB orb = ORB.init (args,null);
      bank mybank = null;
      ...
      try {
         // Bind to any bank object in bank server.
         mybank = bankHelper.bind (":bank");
      }
      catch (SystemException se) {
         // Details omitted.
      }
      ...
   }
}
```

No object name or marker is specified in the `bankHelper.bind()` call, so OrbixWeb chooses any `bank` object within the chosen server. When the `bind()` call is made, the OrbixWeb daemon launches an appropriate process by invoking the Java interpreter on the `javaserver1` class, if a process is not already running.

It is assumed that the `bankHelper.bind()` call binds to one of our newly created `_tie_bank` objects. The result of the binding is an automatically generated proxy object in the client. This acts as a stand-in for the remote `bankImplementation` object in the server. The object reference `mybank` within the client is now a remote object reference as shown in Figure 16 on page 163.

The client programmer is not aware of the TIE object. Nevertheless, all remote operation invocations on the `bankImplementation` object are via the TIE object.

The client program proceeds by asking the bank to open a new account, and making a deposit:

```
// In file javaclient1.java.
// In class javaclient1.
...
try {
   // Obtain a new bank account.
   currAccount = mybank.newAccount ("chris");
   System.out.println ("Account created ");

   // Invoke operations on account.
   currAccount.makeLodgement ((float)56.90);
   System.out.println ("Balance of first account is ");
   System.out.println (""+currAccount.balance());
   mybank.deleteAccount (currAccount);
}
catch (SystemException se) {
   // Details omitted.
}
...
```

When the `mybank.newAccount()` call is made, the method `bankImplementation.newAccount()` is called (via the TIE) within the bank server. This generates a new `accountImplementation` object and associated TIE object. The TIE object is added to the `bankImplementation` object's list of existing `accounts`. Finally, `newAccount()` returns the `account` reference back to the client.

A new proxy is created at the client-side for the `account` object. This is referenced by the `currAccount` variable as shown in Figure 16 on page 163.

If the ImplBase approach is used, the final diagram is as shown in Figure 17 on page 164.

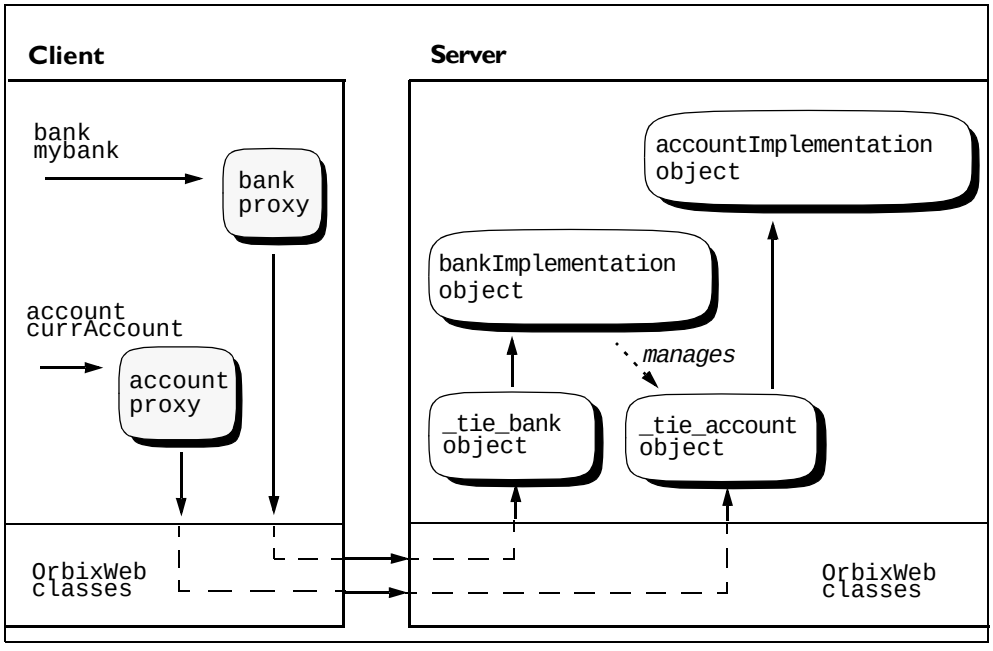

Figure 16: *Client Creates Object (TIE Approach)*

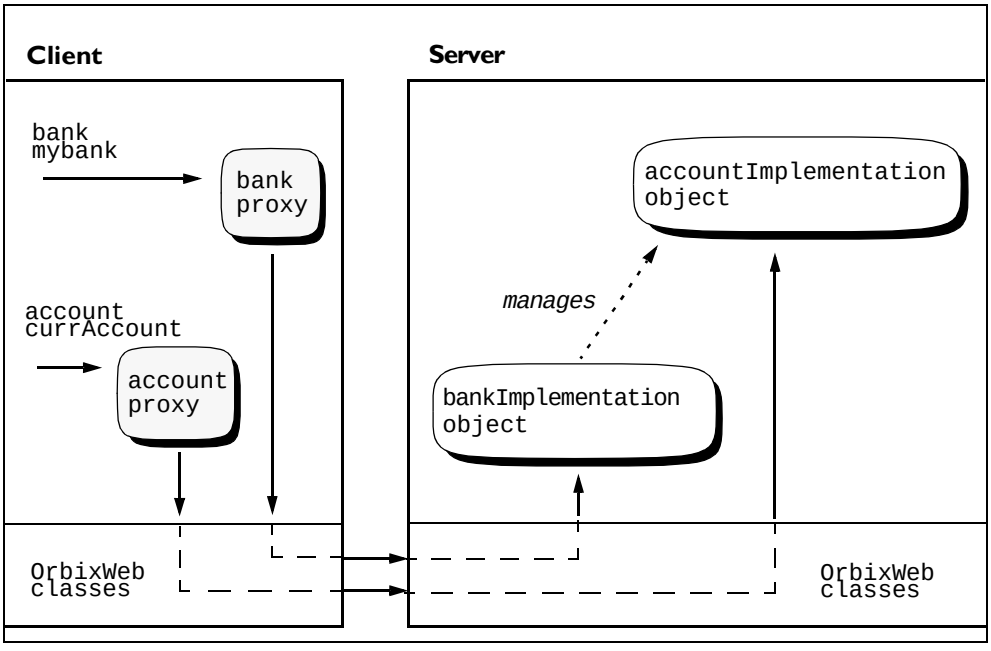

Figure 17: *Client Creates Object (ImplBase Approach)*

Comparison of the ImplBase and TIE Approaches

The TIE and ImplBase approaches to interface implementation impose similar overheads on the implementation programmer. However, there are two significant differences which may affect the choice of implementation strategy:

- The ImplBase approach requires the implementation class to extend a generated base class, while the TIE approach merely requires the implementation of a Java interface.
- The TIE approach requires the creation of an additional object for each implementation object instantiated in a server.

The first of these differences has important implications for the viability of the ImplBase approach in most applications. Java does not support multiple inheritance, so the inheritance requirement which the ImplBase approach imposes on implementation classes limits the flexibility of those classes and eliminates the possibility of reusing existing implementations when implementing derived interfaces. The TIE approach does not suffer from this restriction and, for this reason, is the recommended approach.

The creation of a TIE object for each implementation object may be a significant decision factor in applications where a large number of implementation objects are created and tight restrictions on the usage of virtual memory exist. In addition, the delegation of client invocations by TIE objects implicitly involves an additional Java method invocation for each incoming request.

Of course, it is not necessary to choose one approach exclusively, as both can be used within the same server.

The next two sections examine two important aspects of IDL interface implementation:

- Providing different implementations of the same interface.
- Implementing different interfaces with a single implementation class.

Providing Different Implementations of the Same Interface

Both the ImplBase and TIE approaches allow a you to provide a number of different implementations classes for the same IDL interface. This is an important feature, especially in a large heterogeneous distributed system. An object can then be created as an instance of any one of the implementation classes. Client programmers do not need to be aware of which implementation class is used.

Providing Different Interfaces to the Same Implementation

Using the TIE approach, you can have a Java implementation class which implements more than one IDL interface. This class must implement the generated Java `Operations` interfaces for all of the IDL interfaces it supports, and must therefore implement all the operations defined in those IDL interfaces. This common class is simply instantiated and passed to the constructor of any TIE objects created for a supported IDL interface. This is a way of giving different access privileges to the same object.

With the ImplBase approach, it is not possible to implement different interfaces in a single implementation class, as each interface requires the implementation class to extend an IDL generated base class.

An Example of Using Holder Classes

This section outlines an example of using holder classes in the bank application. You should refer to "Holder Classes and Parameter Passing" on page 100 for a detailed description of holder classes.

Recall the definition of operation `newAccount()`, from interface `bank`, in the IDL definition for the banking application:

```
// IDL
account newAccount(in string name);
```

In order to illustrate the use of Holder types, this IDL definition is modified as follows:

```
// IDL
void newAccount (in string name, out account acc);
```

The IDL compiler maps this operation to a method of Java interface `bank` as follows:

```
// Java
// In package bank_demo.

public void newAccount
   (String name, bank_demo.accountHolder acc);
```

The `out` parameter of type `account` maps to an OrbixWeb `Holder` class, `bank_demo.AccountHolder`, which is generated by the IDL compiler. The source code for this class appears as follows:

```
// Java generated by the OrbixWeb IDL compiler.

package bank_demo;

import IE.Iona.OrbixWeb._OrbixWeb;

public final class accountHolder
   implements org.omg.CORBA.portable.Streamable {

      public bank_demo.account value;
      public accountHolder() {}
      public accountHolder(bank_demo.account value) {
      ...
      }
}
```

This `Holder` class stores a `value` member variable of type `account`, which can be modified during the operation invocation. The example client application can be now coded as follows:

```
// Java
// Alternative client code

package bank_demo;

import org.omg.CORBA.SystemException;

public class javaclient2 {
   public static void main (String args[]) {
      bank mybank = null;
      account currAccount = null;
      accountHolder aHolder = new accountHolder ();
      float f = (float) 0.0;
```

```
        try {
           // Bind to any bank object.
           // in bank server.
           mybank = bankHelper.bind (":bank");

           // Obtain a new bank account.
           mybank.newAccount ("Joe", aHolder);
        }
        catch (SystemException se) {
           System.out.println
              ("Unexpected exception on bind");
           System.out.println (se.toString ());
        }

        // Retrieve value from Holder object.
        currAccount = aHolder.value;

        try {
           // Invoke operations on account.
           currAccount.makeLodgement ((float)56.90);
           f = currAccount.balance();
           System.out.println ("Current balance is "+ f);
        }
        catch (SystemException se) {
           System.out.println
              ("Unexpected exception"
                 + " on makeLodgement or balance");
           System.out.println (se.toString ());
        }
    }
```

In the server, the implementation of method newAccount() receives the Holder object for type account and can manipulate the value field as required. For example, in this case the newAccount() method could instantiate a new account implementation object as follows:

```
// Java
// In class bankImplementation.

public void newAccount
   (String name,bank_demo.accountHolder acc) {
   accountImplementation accImpl =
      new accountImplementation (0, name);

   acc.value = new _tie_account (accImpl);

   ...
}
```

Holder classes are required for all out or inout parameters.

Note: If the account parameter in the IDL definition was labelled inout, you would need to instantiate the value member of the Holder class before calling the newAccount() operation.

For more information on Holder classes refer to Chapter 6, "IDL to Java Mapping" on page 91.

8

Making Objects Available in OrbixWeb

A central requirement in a distributed object system is for clients to be able to locate the objects they wish to use. This chapter describes how object references are published and resolved in OrbixWeb. OrbixWeb Naming Service, IONA's implementation of the CORBA Naming Service, is described in this chapter.

Note: OrbixWeb Naming Service is available with the OrbixWeb Professional Edition.

The role of the CORBA Naming Service is to allow a name to be bound to, or associated with, an object and to allow that object to be found by resolving that name within the Naming Service. The OrbixWeb Naming Service is IONA's implementation of the CORBA Naming Service. This chapter describes in detail how to use the OrbixWeb Naming Service, and briefly describes a couple of alternative methods for making object references available to clients.

OrbixWeb Object References

Every CORBA object is uniquely identified by its object reference. An object reference is an internal identification structure which contains a fixed set of fields. This object reference allows a client to locate the object in the system.

If an application wishes to bind to an object, it must have a mechanism for obtaining the information stored in the object reference. There are several ways for a server to make this information available. These are discussed in "Making Objects Available to Clients" on page 177.

Each OrbixWeb object reference includes the following information:

- An object name that is unique within its server.
  This name is referred to as the object's *marker*.
- The object's server name
  (sometimes called an *implementation name* in CORBA terminology).
- The server's host name.

For example, the object reference for a bank account would include the object's marker name, the name of the server that manages the account, and the name of the server's host. The bank server could, if necessary, create and name different bank objects with different names, all managed by the same server.

In more detail, an OrbixWeb object reference is fully specified by the following fields:

- Object marker.
- Server name.
- Server host name.
- IDL interface type of the object.
- Interface Repository (IFR) server in which the definition of this interface is stored.
- IFR server host.

All OrbixWeb objects implement the Java interface `org.omg.CORBA.Object`. This interface supplies several methods common to all object references including `object_to_string()` which produces a stringified form of the object reference. The form of the resultant string depends on the protocol being used. In the case of IIOP, a string

representation of an IOR is produced. In the case of Orbix Protocol, a string of the following form is produced:

```
:\server_host:server_name:marker:IFR_host:IFR_server:IDL_interface
```

`IE.Iona.OrbixWeb.CORBA.ObjectRef` also provides access to the individual fields of an object reference string via the following set of accessor methods:

```
// Java
// in package IE.Iona.OrbixWeb.CORBA,
// in interface ObjectRef.
public String _host();
public String _implementation();
public String _marker();
public String _interfaceHost();
public String _interfaceImplementation();
public String _interfaceMarker();
```

OrbixWeb automatically assigns the server host, server name and IDL interface fields on object creation and it is not generally necessary to update these values. OrbixWeb also assigns a marker value to each object, but you may choose alternative marker values in order to explicitly name OrbixWeb objects. The assignment of marker names to objects is discussed in the following section. In general, the IFR host name (`interfaceHost`) and IFR server (`interfaceImplementation`) fields are set to default values. In the stringified form these are 'IFR' and the blank string, respectively.

Assigning Markers to OrbixWeb Objects

An OrbixWeb marker value allows a name (in string format) to be associated with an object, as part of its object reference. You can specify a marker name at the time an object is created, as shown in "Assigning a Marker on Creation" on page 174. If you do not specify a marker for a newly created object, a name is automatically chosen by OrbixWeb.

You can use the modifier method `_marker(String)` to rename an object which has a user-specified name or a name assigned by OrbixWeb. This is defined in the interface `ObjectRef`, in package `IE.Iona.OrbixWeb.CORBA`. For details on how to convert an OrbixWeb object to an instance of `ObjectRef`, refer to the class `_OrbixWeb.ObjectRef` in the *OrbixWeb Programmer's Reference*.

You can use the accessor method `_marker()` to find the marker name associated with an object. The following code demonstrates the use of this method:

```
// Java
import org.omg.CORBA.SystemException;
import IE.Iona.OrbixWeb._OrbixWeb;
...

account a;

try {
   a = new _tie_account
      (new accountImplementation ());
   System.out.println ("The marker name chosen " +
   "by OrbixWeb is " + _OrbixWeb.Object(a)._marker ());
}
catch (SystemException se) {
   ...
}
```

Assigning a Marker on Creation

To assign a marker for an object on creation, do either of the following:

- Pass a marker name to the second parameter (of type `String`) of a TIE-class constructor. For example:

```
// Java
import org.omg.CORBA.SystemException;
...

bank b;

try {
   b = new _tie_bank
      (new bankImplementation (), "College_Green");
}
catch (SystemException se) {
   ...
}
```

- Pass a marker name to the first parameter (of type `String`) of a ImplBase class constructor. For example:

```
// Java
// Constructor definition in implementation class:

public class bankImplementation
   extends _bankImplBase {

   bankImplementation (String marker) {
      super (marker);
   }
}

// Usage in server class:
import org.omg.CORBA.SystemException;
...

bankImplementation bankImpl;

try {
   bankImpl = new bankImplementation
      ("College Green");
}
catch (SystemException se) {
   ...
}
```

Choosing Marker Names

The marker names chosen by OrbixWeb consist of a string composed entirely of decimal digits. To ensure that your markers are different from those chosen by OrbixWeb, do not use strings consisting entirely of digits.

Note: Marker names cannot contain the character ':' and cannot contain the *null character*.

An object's interface name together with its marker name must be unique within a server. If a chosen marker is already in use when an object is named, OrbixWeb assigns a different marker to the object. The object with the original marker is not affected. There are two ways to test for this, depending on how a marker is assigned to an object:

- If `IE.Iona.OrbixWeb.CORBA.ObjectRef._marker(String)` is used, you can test for a `false` return value. A `false` return value indicates a name clash.
- If the marker is assigned when calling a TIE-class or an ImplBase class constructor, you can test for a name clash by calling the accessor method `IE.Iona.OrbixWeb.CORBA.ObjectRef.marker()`on the new object and comparing the marker with the one the programmer tried to assign.

Interoperable Object References

OrbixWeb supports two protocols for communications between distributed applications:

- The Orbix protocol.
- The CORBA standard Internet Inter-ORB Protocol (IIOP).

  IIOP enables interoperability between different ORB implementations. This is the default protocol. Refer to Chapter 9, "ORB Interoperability" for a detailed discussion of IIOP.

Both of the two available protocols require a different object reference format. The Orbix protocol requires an OrbixWeb object reference format. IIOP requires the CORBA Interoperable Object Reference (IOR) format. This section introduces object references and shows how you may use the fields of an object reference.

IOR Structure

An object which is accessible via IIOP is identified by an interoperable object reference (IOR). Since an ORB's object reference format is not prescribed by the OMG, the format of an IOR includes the following:

- An ORB's internal object reference.
- An internet host address.
- A port number.

An IOR is managed internally by the ORB. It is not necessary for you to know the structure of an IOR. However, an application may wish to publish the stringified form of an object's IOR. You can obtain the stringified IOR by calling the method `org.omg.CORBA.ORB.object_to_string()` with the required object or `_object_to_string()` on the `IE.Iona.OrbixWeb.CORBA.ObjectRef` interface of the required object.

Making Objects Available to Clients

Clients must be able to locate objects in a distributed system. Consequently, servers must make information about the objects they create available to other system components. There are three fundamental ways of making objects available to clients:

- Using the CORBA Naming Service.

  The OrbixWeb Naming Service is an implementation of the CORBA Naming Service. The Naming Service is the preferred method for locating objects in servers. OrbixTrader may also be used to make objects available, but this is not discussed here.

- Using the `bind` method.

  This method is an OrbixWeb-specific alternative to the Naming Service.

- Using Object Reference Strings to create proxy objects.

The CORBA Naming Service

The CORBA Naming Service holds a 'database' of *bindings* between names and object references. A server that holds an object reference can register it with the Naming Service, giving it a unique name that can be used by other components of the system to locate that object. A name registered in the Naming Service is independent of any properties of the object, such as the objects's interface, server or host name.

The OrbixWeb Naming Service is IONA's implementation of the CORBA Naming Service.

The bind Method

The OrbixWeb-specific `bind()` method provides a mechanism for creating proxies for objects which have been created in servers. A client which uses `bind()` to create a proxy does not need to specify the entire object reference for the target object.

Using Object Reference Strings to Create Proxy Objects

Given a stringified form of an OrbixWeb object reference, an OrbixWeb client can create a proxy for that object, by passing the string to the method `string_to_object()` on an instance of `org.omg.CORBA.ORB`.

The OrbixWeb Naming Service

This section describes the features of OrbixWeb Naming Service, IONA's full implementation of the CORBA Naming Service. The following topics are outlined:

- Terminology and the CosNaming module.
- Format of names within OrbixWeb Naming Service.
- The interfaces `NamingContext` and `BindingIterator`.
- Exceptions raised by operations in OrbixWeb Naming Service.

Terminology and the CosNaming Module

The Naming Service maintains a database of *bindings* between names and object references. A binding is an association between a name and an object. The Naming Service provides operations to resolve a name, to create new bindings, delete existing bindings and to list the bound names.

The interfaces which are provided by the Naming Service are defined within the IDL module `CosNaming`:

```
// IDL
module CosNaming {

   typedef string Istring;
   struct NameComponent {
      Istring id;
      Istring kind;
   };
   typedef sequence<NameComponent> Name;

   enum BindingType {nobject, ncontext};
   struct Binding {
      Name          binding_name;
```

```
    BindingType   binding_type;
};
typedef sequence <Binding> BindingList;

interface BindingIterator;

interface NamingContext {
    enum NotFoundReason {missing_node,
                         not_context, not_object};
    exception NotFound {
        NotFoundReason   why;
        Name             rest_of_name;
    };
    exception CannotProceed {
        NamingContext  cxt;
        Name           rest_of_name;
    };
    exception InvalidName {};
    exception AlreadyBound {};
    exception NotEmpty {};

    void bind(in Name n, in Object obj)
        raises (NotFound, CannotProceed,
                InvalidName, AlreadyBound);
    void rebind(in Name n, in Object obj)
        raises (NotFound, CannotProceed,
                InvalidName);
    void bind_context(in Name n,
                in NamingContext nc)
        raises (NotFound, CannotProceed,
                InvalidName, AlreadyBound);
    void rebind_context(in Name n,
                in NamingContext nc)
        raises (NotFound, CannotProceed,
                InvalidName);
    Object resolve(in Name n)
        raises (NotFound, CannotProceed,
                InvalidName);
    void unbind(in Name n)
        raises (NotFound, CannotProceed,
                InvalidName);
    NamingContext new_context();
    NamingContext bind_new_context(in Name n)
```

```
        raises (NotFound, CannotProceed,
                InvalidName, AlreadyBound);
    void destroy() raises (NotEmpty);
    void list(in unsigned long how_many,
              out BindingList bl,
              out BindingIterator bi);

interface BindingIterator {
    boolean next_one(out Binding b);
    boolean next_n(in unsigned long how_many,
                   out BindingList bl);
    void destroy();
};
```

Format of Names within the Naming Service

A name is always resolved within a given *naming context*. The naming context objects in the system are organised into a naming graph, which may form a naming hierarchy, much like that of a filing system. This gives rise to the notion of a compound name. The first component of a compound name gives the name of a `NamingContext`, in which the second name in the compound name is looked up. This process continues until the last component of the compound name has been reached.

NameComponents

A compound name in the Naming Service takes the more abstract form of an IDL sequence of name components. Also, the name components which make up a sequence to form a name are not simple strings. Instead, a name component is defined as a struct, `NameComponent`, that holds two strings:

```
// IDL
typedef string  Istring;

struct NameComponent {
   Istring id;
   Istring kind;
};
```

The `id` member is intended as the real name component, while the `kind` member is intended to be used by the application layer. For example, you can use the `kind` member to distinguish whether the `id` member should be interpreted as a disk name or a directory

or a folder name. Alternatively, you can use `kind` to describe the type of the object being referred to. The `kind` member is not interpreted by the OrbixWeb Naming Service.

The type `Istring` is a placeholder for a future IDL internationalized string which may be defined by OMG.

A name is defined as a sequence of name components as follows:

```
typedef sequence<NameComponent> Name;
```

Both the `id` and `kind` members of a `NameComponent` are used in name resolution. Thus, two names which differ only in the `kind` member of one `NameComponent` are considered to be different names.

Names with no components (names of length zero) are not allowed.

The NamingContext Interface

The IDL interface `NamingContext` defines the core of the Naming Service.

```
// In module CosNaming.
interface NamingContext {
   // Details shown in this section.
};
```

An application can obtain a reference to its default naming context by passing the string "`NameService`" to the method `resolve_initial_references()` on an instance of `org.omg.CORBA.ORB`:

```
import org.omg.CORBA.ORB;
import org.omg.CORBA.Object;

ORB orb = ORB.init(args,null);
Object initRef = orb.resolve_initial_references("NameService");
```

The result must be narrowed using `CosNaming.NamingContextHelper.narrow()`, to obtain a reference to the naming context.

You can discover which services are available by calling `list_initial_services()`. Refer to *OrbixWeb Programmer's Reference* for details of using this method on `org.omg.CORBA.ORB`.

The NamingContext interface provides operations to:

- Bind a name to an object reference.
- Resolve a name to find an object reference.
- Unbind a name, to remove a binding.
- List the names within a naming context.

These operations are described in the subsections which follow. "Exceptions Raised by Operations in NamingContext" on page 187 describes the exceptions that may be raised by operations defined within interface NamingContext.

All of the operations shown in the following subsections are defined in IDL interface CosNaming::NamingContext.

Resolving Names

Name resolution is the process of looking up a name to obtain an object reference.

resolve()

```
Object resolve(in Name n)
   raises ( NotFound, CannotProceed, InvalidName );
```

The resolve() operation returns the object reference bound to the specified name, relative to the target naming context. This is the naming context on which the operation is invoked. The first component of the specified name is resolved in the target naming context.

The return type is of the resolve() operation is IDL Object. This translates to type org.omg.CORBA.Object in Java. This result must therefore be narrowed, using the appropriate narrow() method, before it can be properly used by an application.

Binding

bind()

```
void bind(in Name n,  in Object o)
   raises ( NotFound, CannotProceed,  InvalidName, AlreadyBound );
```

The bind() operation creates a binding (relative to the target naming context) between a name and an object.

If the name passed to `bind()` is a compound name with more than one component, all name components, with the exception of the last component, are used to find the naming context to which to add the binding. Note that these naming contexts must already exist. The last name component names the specified object reference in the desired naming context.

The `bind()` operation raises an exception if the specified name is already bound within the final naming context.

bind_context()

```
void bind_context(in Name n, in NamingContext nc)
   raises ( NotFound, CannotProceed, InvalidName, AlreadyBound );
```

The `bind_context()` operation creates a binding between a name and a specified naming context, relative to the target naming context. This new binding may be used in any subsequent name resolutions. The entries in naming context `nc` may be resolved using compound names.

All but the final naming context specified in parameter `n` must already exist. This operation raises an exception if the name specified by `n` is already in use.

Note that the naming graph built using `bind_context()` is not restricted to being a tree. This can be a general naming graph in which any naming context can appear in any other.

It is also possible to create a binding between a name and a naming context using `bind()`. This is because interface `NameContext` is a derived interface of interface `Object`. An object reference to an object of type `NamingContext` can be passed as the second parameter to `bind()`. However, the resulting binding cannot be used as part of a compound name. Only bindings created with `bind_context()` and `rebind_context()` can be used as part of a compound name.

rebind()

```
void rebind(in Name n,  in Object o)
   raises ( NotFound, CannotProceed, InvalidName );
```

The `rebind()` operation creates a binding between a name that is already bound in the context and an object. The previous name is unbound and the new binding is made in its place. As is the case with `bind()`, all but the last component of a compound name must name an existing `NamingContext`. A `NotFound` exception is thrown if the name is not already in use.

rebind_context()

```
void rebind_context(in Name n, in NamingContext nc)
   raises ( NotFound, CannotProceed, InvalidName );
```

The `rebind_context()` operation creates a binding between a name that is already bound in the context and a naming context, `nc`. The previous name is unbound and the new binding is made in its place. As is the case for `bind_context()`, all but the last component of a compound name must name an existing `NamingContext`. A `NotFound` exception is thrown if the name is not already in use.

You can also change a binding between a name and a naming context using `rebind()`. This is because interface `NameContext` is a derived interface of interface `Object`. An object reference to an object of type `NamingContext` can be passed as the second parameter to `rebind()`. However, the resulting binding cannot be used as part of a compound name. Only bindings made with `bind_context()` and `rebind_context()` can be used as part of a compound name.

Deleting a Binding

unbind()

```
void unbind(in Name n)
   raises ( NotFound, CannotProceed, InvalidName );
```

The operation `unbind()` removes the binding between the specified name and object.

Creating Naming Contexts

Two operations are provided to create naming contexts.

new_context()

```
NamingContext new_context()
```

The operation `new_context()` creates a new naming context only, without entering it into the naming graph and without binding it to any name. The returned naming context can subsequently be entered into the naming graph using `bind_context()` or `rebind_context()`. The returned context is created within the same name server as the context that is the target of the call.

bind_new_context()

```
NamingContext bind_new_context(in Name n)
   raises ( NotFound, CannotProceed, InvalidName, AlreadyBound );
```

The operation `bind_new_context()` creates a new naming context and binds it using the specified name, relative to the target naming context. The operation `bind_new_context()` is equivalent to calling `new_context()` followed by `bind_context()`. The returned context is created within the same name server as the context that is the target of the call.

Deleting Contexts

destroy()

```
void destroy()
   raises (NotEmpty);
```

The operation `destroy()` deletes the naming context on which it is invoked. The target naming context must be empty; it must not contain any bindings.

Listing a Naming Context

Before describing the `list()` operation on a `NamingContext`, the different types of bindings must be explained.

Types of Bindings

The operations `bind()`, `rebind()`, `bind_context()` and `rebind_context()` create bindings. However it can be seen from the previous sections that the first two create different forms of binding than the last two. The methods `bind()` and `rebind()` allow a name to be bound to any object, including a `NamingContext`, while `bind_context()` and `rebind_context()` are used to construct the naming network supported by the Naming Service.

The two binding types are captured by the following IDL types:

```
// In IDL module CosNaming.

enum BindingType { nobject, ncontext };
```

```
struct Binding {
   Name binding_name;
   BindingType binding_type;
};
```

When browsing a network of naming contexts, an application can list a `NamingContext` and determine the type of each binding in it.

The operations `bind_context()` and `rebind_context()` create bindings of type `ncontext`; the operations `bind()` and `rebind()` create bindings of type `nobject`.

The important difference is that a binding of type `nobject` cannot be used in a compound name, except as the last element in that name. To draw from familiarity with a filing system, you can view bindings of type `ncontext` as naming "directories", while those of type `nobject` name "files".

Listing Names

```
// In IDL module CosNaming.

typedef sequence<Binding> BindingList;

interface BindingIterator;

interface NamingContext {
   ...
   void list(in unsigned long how_many,[1]out BindingList bl,
      out BindingIterator bi);
};
```

list()

The operation `list()` obtains a list of the name bindings in the target naming context.

The parameter `how_many` specifies the maximum number of bindings that should be returned in the `BindingList` parameter `bl`. The `BindingList` parameter is a sequence of `Binding` structs where each `Binding` indicates the name and type of the binding. The type indicates whether the name is that of an object, possibly a `NamingContext` object or whether it is a name of a node in the naming graph which participates in name resolution.

1. The three parameters used here are an example of a pattern used often in the CORBAservices. They allow a sequence of manageable size to be returned to a client; and the entries that would not fit into that sequence to be obtained using an *iterator*.

If the naming context contains more than the number of requested bindings in the `how_many` parameter, the `list()` operation returns a `BindingIterator`. The number of remaining bindings are given in the parameter `bi`, while the first `how_many` bindings are in parameter `bl`. If the naming context does not contain any additional bindings, the parameter `bi` is a nil object reference.

Exceptions Raised by Operations in NamingContext

The exceptions in `NamingContext` are defined as follows:

```
// In IDL module CosNaming.
interface NamingContext {
   enum NotFoundReason { missing_node,
      not_context, not_object };

   exception NotFound {
      NotFoundReason  why;
      Name  rest_of_name;
   };

   exception CannotProceed {
      NamingContext  cxt;
      Name  rest_of_name;
   };

   exception InvalidName {};
   exception AlreadyBound {};
   exception NotEmpty {};
   . . . .
};
```

Refer to *OrbixWeb Programmer's Reference* for a full listing of the exceptions in `NamingContext`.

These exceptions are raised under the following conditions:

`NotFound`	Indicates that some component of the specified name is not bound. To aid debugging, the remainder of the name is returned in the exception. The first component in the returned name is the component that failed.

`CannotProceed`	Indicates that the Naming Service cannot continue with the operation request for some reason. The OrbixWeb Naming Service does not raise this exception currently.
`InvalidName`	Indicates that the specified name is invalid. A `Name` of length zero (without any name components) is invalid. A `Name` which contains a `NameComponent` whose `id` member is zero or is an empty string is also invalid.
`AlreadyBound`	Indicates that an object is already bound to the specified name. At any time, only one object can be bound to a given name in a naming context.
`NotEmpty`	Indicates that the target naming context contains at least one binding. A naming context cannot be destroyed if it contains any bindings.

The BindingIterator Interface

Recall that interface `NamingContext` provides an operation `list()` to obtain the list of name bindings within a context:

```
// In IDL interface CosNaming::NamingContext.
void list(in unsigned long how_many, out BindingList bl,
   out BindingIterator bi);
```

This operation returns a maximum of `how_many` bindings in the parameter `bl`. If the target context contains more than `how_many` bindings, the `BindingIterator` parameter can be used to access the remaining entries. The relevant IDL definitions are as follows:

```
// In IDL module CosNaming.
enum BindingType { nobject, ncontext };

struct Binding {
   Name binding_name;
   BindingType binding_type;
};

typedef sequence<Binding> BindingList;
```

```
interface BindingIterator {
   boolean next_one(out Binding b);

   boolean next_n(in unsigned long how_many, out BindingList bl);

   void destroy();
};
```

The operations `next_one()` and `next_n()` can be used to access the additional entries. These are the entries which are other than those returned by the `out BindingList bl` parameter in the `list()` operation. Each entry is returned at most once. Hence, you can use consecutive calls to `next_one()` and / or `next_n()` to retrieve additional entries in a naming context. The operation `next_n()` returns at most `n` entries, but can also return less than `n` entries.

The operation `next_one()` returns `true` if an entry can be returned, otherwise it returns `false`.

The operation `next_n()` returns `true` if `n` (or less) entries can be returned. If no entries can be returned, `next_n()` returns `false`.

You can delete a `BindingIterator` object by calling its `destroy()` operation.

Using OrbixWeb Naming Service

This section explains how to use OrbixWeb Naming Service. The example code used here reflects the OMG Standard IDL to Java mapping.

The following topics are discussed:

- Using the `CosNaming` interfaces to create bindings and resolve bound names.
- Building an application using OrbixWeb Naming Service.
- Using the command-line interface to OrbixWeb Naming Service which allows naming context graphs to be built and manipulated.
- Federation of name spaces.

String Format of Names

The convention used for the string representation of names is illustrated by the following example:

```
documents-dir.reports-dir.april98-txt
```

In this example, the first name component's id is `documents` and its kind is `dir`. This is followed by a second component with id `reports` and kind `dir`, which in turn is followed by a component with id `april98` and kind `txt`.

You should include the dash '-' character in every name component. If you omit this, the utilities assume that the `id` and `kind` field are identical. Therefore:

```
documents.reports.april98
```

is synonymous with the following name:

```
documents-documents.reports-reports.april98-april98
```

If you want to omit a kind field from the initial components of a name, finish each component with a dash '-'. An example is `documents-.reports-.april98-txt`. This practice is followed here.

OrbixWeb Naming Service Example

Consider a software engineering company that maintains an administrative database of personnel records which includes details of names, login names, addresses, salary and holiday entitlements. These records are used for various administrative purposes. The Naming Service is used to locate an employee record by name. Figure 18 on page 191 shows part of a naming context graph designed for this purpose.

The nodes `company`, `staff`, `engineering` and `support` represent naming contexts. A name such as `company-.staff-.paula-record` names an object. The same object may have more than one name. For example, each person is listed in both the generic `company-.staff-` context, and is also listed in a particular division such as `company-.engineering-` or `company-.sales-`.

In addition, in this case it is convenient to use 'abstract' names, so that, for example, the person who is engineering manager can be found by looking up the name `company-.engineering-.manager-`.

Allowing different paths to the same data allows many uses to be made of the Naming Service. For example, a payroll system might be interested only in the `company-.staff-`

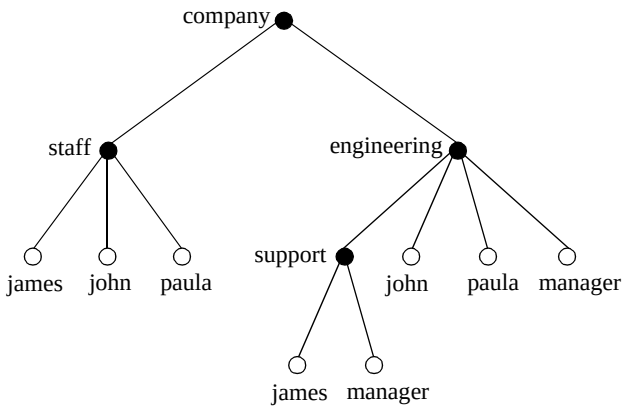

Figure 18: *A Naming Context Graph*

context; while the engineering manager might want the holiday records for all the employees with entries in the `company-.engineering-` context to be written to a spreadsheet.

The remainder of this section shows code samples based on a `Staff` naming context. The full code for a version of this example is available in the `demos/namesStaff` directory of your OrbixWeb installation.

Finding the Default Naming Context

There are three ways for an application to find its initial naming context:

- Using the CORBA Initialization Service.
- Using the `bind()` call.
- Using the root naming context IOR.

Using the CORBA Initialization Service

The CORBA defined and recommended approach is to use the CORBA Initialization Service. To use the Initialization Service, pass the string "NameService" to the following method call on the org.omg.CORBA.ORB:

```
// Java
// In class org.omg.CORBA.ORB.
org.omg.CORBA.Object resolve_initial_references
                                (String identifier)
```

The result must be narrowed, using CosNaming.NamingContextHelper.narrow() to obtain a reference to the naming context.

Note: Using resolve_initial_references() requires that the default Naming Service host and port are configured in the file OrbixWeb.properties. You can set these values using the Initialization page of the Configuration Tool.

Using the bind() Call

A second approach is to use the OrbixWeb specific bind() call. The normal way to do this is to bind to the root naming context in a name server. For example, the following call binds to a naming context with the marker name root, within the server NS:

```
// Java
CosNaming.NamingContextrootContext
      = CosNaming.NamingContextHelper.bind
                            ("root:NS",host);
```

Using the Root Naming Context IOR

A third approach is to read the root naming context IOR from a shared file. You should start up the naming server using the following form of command:

```
ns -I /sharedIORs/ns.ior
```

The IOR for the root naming context is stored in this file as the naming server starts up. You can use this IOR to obtain the initial naming context:

```
// Java
import org.omg.CORBA.ORB;
      ...
```

```
String rootIOR;
org.omg.CORBA.Object objRef;
ORB orb = ORB.init(args, null);
// Read the contents of file /sharedIORs/ns.ior
// into the string rootIOR...
   ...
objRef = orb.string_to_object(rootIOR);
```

The resulting object reference must subsequently be narrowed using the following call:

`CosNaming.NamingContextHelper.narrow(objRef).`

Once a program has a reference to the initial naming context, it can look up further names in contexts held in that name server. Other naming servers can also be used, because it is possible for a name in one naming server to name a naming context in another naming server. Refer to "Federation of Name Spaces" on page 203 for more details.

Creating a Naming Context

The code in this section shows how to build a `Staff` naming context. The following IDL interface is assumed:

```
// IDL
interface Person {
   readonly attribute string name;
   readonly attribute string home_address;
   readonly attribute string job_title;
   readonly attribute string user_id;
   readonly attribute string phone_extn;
   // More employee related information.
};
```

In the example the `Person` IDL definition is implemented by a `PersonImplementation` class.

The following server code resolves the `Staff` context in the root context. If the `Staff` context does not exist, it is created. Exception handling is not included:

```
// Java
// An OrbixWeb server(and client of the Naming
// Service).

import org.omg.CORBA.ORB;
import org.omg.CosNaming.*
```

```
...

public class javaserver1 {

   static NamingContext rootContext = null;
   static NamingContext staffContext = null;
   static org.omg.CORBA.ORB orb = null;
   public static void main (String args[]) {

      orb = ORB.init (args,null);
      ...

      // find the initial naming context
      try {
         org.omg.CORBA.Object initNCRef
            = orb.resolve_initial_references
                                       ("NameService");
         rootContext = NamingContextHelper.narrow
                                         (initNCRef);
      }
      catch() {}
      // catch clause not implemented here

      PersonImplementation john = null;
      PersonImplementation colm = null;
      PersonImplementation paula = null;

      try {
         john = new PersonImplementation
                              ("John","Architect");
         colm = new PersonImplementation
                              ("Colm","Engineer");
         paula = new PersonImplementation
                              ("Paula","Manager");
      }
      catch() {}
      // catch clause not implemented here

      // A NameComponent[] is an array of structs
      NameComponent[] name = new NameComponent[2];
      NameComponent[] NC = new NameComponent[1];
      NC[0]  = new NameComponent ("Staff","Staff");
```

```
System.out.println ("Resolving...");

//resolve the "Staff" context in the root context
try {
   rootContext.resolve (NC);
}
catch() {}
// catch clause not implemented here

System.out.println ("Bind new context...");
...

// if "Staff" does not exist then create it
try {
   staffContext
       = rootContext.bind_new_context(NC);
   System.out.println
      ("Created a new context Staff in the
                              root context\n");
}
...
```

Binding Names

The server code samples in this section show two different methods of binding names to the `Staff` naming context:

1. The first method involves binding a name from the `root` context, as follows:

```
// Java
// Server code extract

   name[0] = new NameComponent ("Staff","Staff");
   name[1] = new NameComponent ("john","john");
   rootContext.bind (name,john);

   System.out.println
      ("Bound Object"
         +_OrbixWeb.Object(john)._object_to_string()
              + "to name john in context root.staff");
```

2. The second method involves binding a name from the `Staff` context, as follows:

```
// Java
// Server code extract

name[1] = new NameComponent ("paula","paula");
System.out.println ("Binding to name@paula");
rootContext.bind (name, paula);
System.out.println
   ("Bound Object "
      + _OrbixWeb.Object(paula)._object_to_string()
           + " to name paula in context root.staff");
```

NameComponent Arrays

Recall from "Format of Names within the Naming Service" on page 180 that a `NameComponent` is defined as follows:

```
public class NameComponent {
   String id;    // Context/Object ID
   String kind;  // Context/Object description
}
```

In order to retrieve an object reference bound to a name, you must construct a `NameComponent` array. This array takes the following form:

```
contextID-contextKIND.contextID-contextKIND.objectID-objectKIND
```

The last element of this array represents the required object. This array is then passed as a parameter to `resolve()` which is called on the required context, as shown in the next section.

Resolving Names

For a client, a typical use of the Naming Service is to find the initial naming context; and then to resolve a name, to obtain the object reference that is bound to the specified name. This is illustrated in the code segments which follow. The client finds the object named `Staff.john` and then prints out its details.

The client is written as follows:

```
// Java
// An OrbixWeb client

import org.omg.CORBA.ORB;
import IE.Iona.OrbixWeb.CosNaming.*;
...

public class javaclient1 {

   static NamingContext rootContext, staffContext
                                          = null;
   static namesStaff.Person personRef = null;
   static org.omg.CORBA.ORB orb = null;

   public static void main( String[] args ) {
      ....
      NamingContext rootContext = null;

      orb = ORB.init (args,null);

      // find intial naming context
      try {
         org.omg.CORBA.Object initNCRef
              = orb.resolve_initial_references
                                    ("NameService");
         rootContext = NamingContextHelper.narrow
                                    (initNCRef);
       }
      catch() {}
      // catch clause not implemented here
      ...
```

The following are two methods of retrieving a object reference which is bound to a specified name:

1. The first methods involves calling `resolve()` on the `root` context. In this example, the context is `Staff` and the required object is `john`:

```
// Java
// Client code extract

// Resolve name 'Staff.john'
NameComponent[] name = new NameComponent[2];
org.omg.CORBA.Object objRef = null;

name[0] = new NameComponent ("Staff","Staff");
name[1] = new NameComponent ("john","john");
objRef = rootContext.resolve (name);
personRef = namesStaff.PersonHelper.narrow (objRef);
printDetails (personRef);
```

2. The second method is a two step process. This involves retrieving the `Staff` context from the `root` context and resolving the object `john` from the `root` context, as follows:

```
// Java
// Client code extract

//Resolve name 'Staff.colm'
name = new NameComponent[1];
org.omg.CORBA.Object staffCtxRef = null;
name[0] =  new NameComponent ("Staff", "Staff");

staffCtxRef = rootContext.resolve (name);

name[0] =  new NameComponent ("colm","colm");

NamingContext staffCtx
   = NamingContextHelper.narrow (staffCtxRef);
objRef = staffCtx.resolve (name);
personRef = namesStaff.PersonHelper.narrow (objRef);
printDetails (personRef);
```

Listing Context Bindings

The following client code extract shows a simple example of using the `BindingIterator` interface to list the bindings in the Staff context:

```
// Java
// Client code extract

// List all the staff context:
BindingListHolder blist
                      = new BindingListHolder () ;
BindingIteratorHolder biterHolder
                      = new BindingIteratorHolder ();
BindingHolder binding = new BindingHolder ();

NameComponent[] NC = new NameComponent[1];
NC[0] = new NameComponent ("Staff", "Staff");
objRef = rootContext.resolve (NC);
staffContext = NamingContextHelper.narrow (objRef);

//Deliberately make the "how_many" argument too small,
//only 2 out of 3 names will be returned.

staffContext.list (2,blist,biterHolder);
System.out.println
   ("\nContents of staff context:");
System.out.println
   ("The length of the list is " + blist.value.length);
System.out.println
     (blist.value[0].binding_name[0].id);
System.out.println
   (blist.value[1].binding_name[0].id);
System.out.println
   ("\nPrint the remaining objects");

// print the remaining objects
if (biterHolder.value != null ) {
while ( biterHolder.value.next_one (binding))
   System.out.println
     (binding.value.binding_name[0].id);
```

Compiling and Running a Naming Service Application

This section outlines how to build a demonstration program that uses the Naming Service. It describes what configuration variables are required, how to register a naming server in the Implementation Repository and what options are available on the naming server executable.

Building the OrbixWeb Naming Service Demonstration Application

The Naming Service demonstration program is located in the `\demos\namesStaff` directory of your OrbixWeb installation.

Use the following steps for running the demonstration application:

1. To build the application on Solaris use `gmake`; on Windows run the `compile.bat` batch program.
2. Register the Naming Service by entering the following command:
   ```
   putit -j NS IE.Iona.OrbixWeb.CosNaming.NS
   ```
3. Register the `Staff` server by entering the following command:
   ```
   putit -j Staff namesStaff.javaserver1
   ```
4. Start the Java server by running the `javaserver1` script on Solaris or `javaserver1.bat` on Windows. This launches the Naming Service and populates it with names.
5. Start the Java client by running the `javaclient1` script on Solaris or `javaclient1.bat` on platforms. This establishes a connection with the Naming Service and resolves the names bound by the Java server.

Configuring OrbixWeb Naming Service

A default configuration for OrbixWeb Naming Service is set up at installation. The configuration parameters are stored in the `OrbixWeb.properties` file. These comprise the following entries:

Variable	Description
`IT_NAMES_REPOSITORY_PATH`	This represents the default location of the Naming Service Repository entries. By default, this is set to the following directory: `<install dir>/configuration/NamingRepository`
`IT_NAMES_TIMEOUT`	The default timeout, set to the following: `-1(IT_INFINITE_TIMEOUT)`
`IT_NAMES_HASH_TABLE_SIZE`	The initial size for the Naming Service hash table This value must be a prime number.
`IT_NAMES_HASH_TABLE_LOAD_FACTOR`	Percentage of table elements used before a resize.

`Client configuration:`
`IT_NAMES_SERVER`
`IT_NS_PORT`
`IT_NS_HOSTNAME`
`IT_NS_IP_ADDR`
	By default, a call to `resolve_initial_references ("NameService")` from the ORB expects a naming server to be registered in the Implementation Repository with the name "`NS`".
	If these variables are set, `resolve_initial_references()` searches for a naming server with the name specified on the specified host and port.
	If the configuration parameter `IT_INITIAL_REFERENCES` contains a naming service reference, this overrides the above.

Table 3: *OrbixWeb Naming Service Configuration*

Registering a Name Server with the Implementation Repository

As a normal OrbixWeb server, a name server must be registered with the OrbixWeb Implementation Repository.

As usual, register a name server using the `putit` utility as appropriate to the target platform. Using `putit`, a typical command to register a name server is as follows:

```
putit -java NS IE.Iona.OrbixWeb.CosNaming.NS
```

Once registered with the Implementation Repository, the name server can either be activated by the OrbixWeb daemon or be manually launched.

You can terminate the name server in the same way as any OrbixWeb server: using the `killit` utility on UNIX, or using the Server Manager utility on Windows.

Options to the Name Server

The OrbixWeb name server executable is named `ns`. This takes the following options:

```
ns [-v] [-t <timeout>] [-I <ns-ior-file>] [-r <repository]
```

The options are listed as follows:

`-I <ns-ior-file>`	Specifies a file where the name server stores the root context IOR when it starts up.
`-r <repository>`	Specifies the location of the Naming Service Repository
`-t <timeout>`	Specifies the period of time, in milliseconds, that the name server may remain idle before timing out. The default timeout is infinite, meaning that the name server does not time out.
`-v`	Outputs version information. Specifying `-v` does not cause the name server to be run.

Federation of Name Spaces

The collection of all valid names recognised by the Naming Service is called a *name space*. A name space is not necessarily located on a single name server: a context in one name server can be bound to a context in another name server on the same host or on a different host. The name space provided by a Naming Service is the association or *federation* of the name spaces of each individual name server that comprises the Naming Service.

Figure 19 on page 203 shows a Naming Service federation that comprises two name servers running on different hosts. In this example, names relating to the company's engineering and PR divisions are located on one server and names relating to the company's marketing division are located on a separate server. Client requests to look up names start in one name server but may continue in another name server's database. Clients do not have to be aware that more than one name server is involved in the resolution of a name, and they do not need to know which server interprets which part of a compound name.

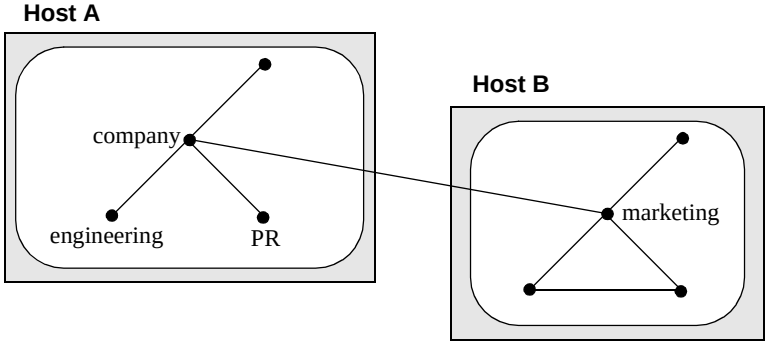

Figure 19: *Naming Graph Spanning Different Name Servers*

Binding to Objects in OrbixWeb Servers

Note: This section discusses the use of the OrbixWeb `bind()` method to create proxy objects in clients. This should not be confused with the use of `bind()` in the context of OrbixWeb Naming Service.

There is a difference between binding to OrbixWeb servers and binding in a Naming Service. Binding in a Naming Service context involves associating an application level name, usually a meaningful string, to an IOR. This binding is used at resolution time to map a name to an object through its IOR. Binding to servers, however, involves the creation of a proxy object in the client through which methods on the remote server may be activated.

The OrbixWeb `bind()` method provides a mechanism for creating proxies for objects that have been created in servers. A client that uses `bind()` to create a proxy does not need to specify the entire object reference for the target object. Although `bind()` can be invoked using either the Orbix protocol or CORBA IIOP, it can only succeed if the target object is implemented in an Orbix or OrbixWeb server. The `bind()` method cannot be used with objects which are implemented using other ORBs.

The creation of a proxy in a client's address space allows you to invoke operations on the target object. When an operation is invoked on the proxy, OrbixWeb automatically transmits the request to the target object. You can use the `bind()` method to specify the exact object required or, by using default parameters, OrbixWeb is allowed some freedom when choosing the object.

In OrbixWeb, the `bind()` method has been integrated with a locator which provides a basic service for finding objects when no host is specified. Refer to "Locating Servers at Runtime" on page 473 for more details.

The bind() Method

The `bind()` method is a static method automatically generated by the IDL compiler for each IDL Java class. For interface `account`, the full form of `bind()` is declared in Java class `accountHelper` as follows:

```
// Java
public static final accountHelper
   bind(String markerServer, String host) {
      ...
}
```

The bind() method is overloaded and takes the following sets of parameters:

- No parameters
- orb
- markerServer
- markerServer, orb
- markerServer, host
- markerServer, host, orb
- A full object reference as returned by the method org.omg.CORBA.ORB.object_to_string().

The bind() method supports polymorphic binds. This means that you can make a call to accountHelper.bind() to an object of interface currentAccount, if interface account is a base interface of interface currentAccount.

The orb parameter enables support for multiple ORBs. The specific ORB passed to the bind() method is used to build the proxy and establish a connection to the target server when required. The markerServer and host parameters are explained in turn in the following pages. Table 4 on page 209 contains a tabular summary of the parameters to bind(), and "Binding and Exceptions" on page 210 describes how bind() can raise an exception and how to optimize its performance.

Finally, this chapter ends with a description of methods of creating proxy objects from object reference information, including binding to a stringified object reference.

The markerServer Parameter to bind()

The markerServer parameter denotes both a specific server name and object within that server. It can be a string of the following form:

```
marker : server_name
```

The marker identifies a specific object within the specified server. The server_name is the name of a server, as registered in the OrbixWeb Implementation Repository. It is not necessarily the name of a class or an interface although you can give a server the same name as that of a class or interface. The OrbixWeb Implementation Repository is described in detail in "Registration and Activation of Servers" on page 251.

If the server name is not given in the markerServer parameter, the server name defaults to the name of the Java class for bind().[2] For example, in a parameterless call to

bankHelper.bind(), the server name defaults to "bank".[3] This means that the target server must have been registered with the name "bank".

If the marker is not given, it defaults to any object within the server that implements the interface, or derived interface, given by the specified Java class name. The chosen object may have been named explicitly by the programmer or assigned a default name by OrbixWeb.

If the string does not contain a ':' character, the string is understood to be a marker with no explicit server name. Because a colon is used as the separator, it is invalid for a marker or a server name to include a ':' character.

The following are examples of the markerServer parameter which could be used in a call to bankHelper.bind():

"College_Green:AIB"	The College_Green object at the AIB server.
"College_Green"	The College_Green object at the bank server.
"College_Green:"	The College_Green object at the bank server.
""	Any bank object at the bank server.
"College_Green:mybank"	The College_Green object at the mybank server.
":mybank"	Any bank object at the mybank server.

Finally, if the markerServer parameter contains at least *two* ':' characters, it is not treated as a marker:server_name pair. However, it is assumed to be the string form of a full *object reference*. Refer to "Using Object Reference Strings to Create Proxy Objects" on page 211 for more details.

2. This is the only system recognised relationship between server names and interface names.

3. OrbixWeb will choose the name of the Java class if a null string is specified for the server name. You can do this either by not passing a first parameter, or by passing one of the following as the first parameter: a null string; a string with no ':'; or a string which terminates with a ':'.

The host Parameter to bind()

The `host` parameter to `bind()` specifies the Internet host name or the Internet address of a node on which to find the object. An Internet address is assumed to be a string of the form `xxx.xxx.xxx.xxx`, where `x` is a decimal digit.

Where a null string is provided, OrbixWeb uses the *default locator* to find the object's server in the distributed system. OrbixWeb's default locator allows the locations of servers to be recorded, as is explained in "Locating Servers at Runtime" on page 473. This configuration information is then used during `bind()`, provided that the host parameter is not explicitly given. The locator must be configured before it can be used.

OrbixWeb also allows you to override the default locator with an alternative location mechanism. The programming steps required to achieve this are described in "Locating Servers at Runtime" on page 473.

Example Calls to bind()

This section shows a selection of sample calls to `bind()`.

1. Bind to any `bank` object in any bank server. That object should implement the bank IDL interface.

   ```
   bank b = bankHelper.bind();
   ```

2. Bind to any `bank` object in the bank server at node `alpha` (in the current domain). That object should implement the bank IDL interface.

   ```
   bank b = bankHelper.bind("", "alpha");
   ```

3. Bind to the `College_Green` object within the bank server at node `alpha` (in the current domain). That object should implement the bank IDL interface.

   ```
   bank b = bankHelper.bind("College_Green", "alpha");
   ```

Note: It is generally recommended that you include the server name, if known. It is generally not recommended to use the IDL interface name as the server name.

4. Bind to the College_Green object (in server bank) somewhere within the network. College_Green should implement the bank IDL interface.

```
bank b = bankHelper.bind("College_Green");
```

5. Bind to the College_Green object in the AIB server somewhere in the network. That object must implement the bank IDL interface.

```
bank b = bankHelper.bind("College_Green:AIB");
```

6. Bind to the College_Green object at the AIB server at node beta, in the internet domain mc.ie. That object should implement the bank IDL interface.

```
bank b = bankHelper.bind
   ("College_Green:AIB", "beta.mc.ie");
```

7. Bind to the College_Green object at the AIB server at Internet address 123.456.789.012. That object should implement the bank IDL interface.

```
bank b = bankHelper.bind
   ("College_Green:AIB", "123.456.789.012");
```

Tabular Summary of bind() Parameters

Table 4 summarises the rules for a general form call to `bind()`:

```
// Java
T1 t;
t = T2Helper.bind("M:S", "H", "O");
```

`T1`	`T1` must be the same or a base type of `T2`.
`T2`	`T2` is an IDL interface name, and also a Java interface name. It is not the name of a server, unless a server is explicitly created with the same name. The object that is found must implement interface `T2` or a derived interface of this.
`M`	`M` is a marker name. This is the name of an object within the specified server. If `M` is left blank (if the `markerServer` parameter to `bind()` is the empty string, or begins with a "`:`" character), `bind()` is allowed to find any object in the specified server with a correct interface (`T2` or a derived interface).
`S`	`S` is a server name, used previously to register a server in the Implementation Repository. If `S` is left blank (if the `markerServer` parameter to `bind()` is the empty string, or has no "`:`" character, or terminates with a "`:`" character), the name `T2` is used as the server name. In this case, a server must have been explicitly registered with the name `T2`.
`H`	`H` is an Internet host name or (if the string is in the format `xxx.xxx.xxx.xxx`, where `x` is a decimal digit) an Internet address. If `H` is the empty string, OrbixWeb uses its *locator* to try to find the required server.
`O`	`O` is an additional optional `org.omg.CORBA.ORB` which provides support for multiple ORBs. The supplied `ORB` is used to build the proxy and establish a connection to the target server when required.

Table 4: *Summary of Parameters to bind()*

To bind to an object with interface T2 and marker "aaa" in a server called "sss", known to be running on host "hhh", you could write (ignoring exception handling):

```
// Java
T2 tRef;
tRef = T2Helper.bind("aaa:sss","hhh");
```

Since a server can support objects of any number of interfaces, the following can be used to bind to an object of interface T3 and marker "bbb" in the same server:

```
// Java
T3 tRef;
tRef = T3Helper.bind("bbb:sss","hhh");
```

Binding and Exceptions

By default, `bind()` raises an exception if the desired object is unknown to OrbixWeb. This requires OrbixWeb to ping the desired object in order to check its availability The ping operation is defined by OrbixWeb and has no effect on the target object. The pinging causes the target OrbixWeb server process to be activated if necessary, and confirms that this server recognises the target object.

If you wish to improve efficiency by reducing the number of remote invocations, pinging can be disabled by calling the method `pingDuringBind()` as follows:

```
// Java
import IE.Iona.OrbixWeb._CORBA;
...
_CORBA.Orbix.pingDuringBind(false);
```

When pinging is disabled, binding to an unavailable object does not raise an exception at that time. Instead, an exception is raised when the proxy object is first used.

A program should always check for exceptions when calling `bind()`, whether or not pinging is enabled. Even when pinging is disabled, this method raises an exception in some circumstances, including on some configuration errors.

Using Object Reference Strings to Create Proxy Objects

An OrbixWeb object is uniquely identified by an object reference. Given a stringified form of an OrbixWeb object reference, an OrbixWeb client can create a proxy for that object, by passing the string to the method `string_to_object()` on an instance of `org.omg.CORBA.ORB`.

For example, given an object reference string which identifies a `bank` object:

```
// Java
import org.omg.CORBA.ORB;
import org.omg.CORBA.Object;
import org.omg.CORBA.SystemException;
import IE.Iona.OrbixWeb._CORBA;
...

//Assign to object ref string.
String bStr = ... ;
bank b;

ORB orb = ORB.init(args, null);

try {
     Object o = orb.string_to_object ( bStr );
     b = bankHelper.narrow ( o );
}
catch (SystemException se) {
   ...
}
```

Similarly, the `markerServer` field of the `bind()` method can accept a stringified object reference:

```
// Java
import org.omg.CORBA.SystemException;
...

// Assign to object ref string.
String bStr = ...;
bank b;

try {
   b = bankHelper.bind (bStr);
}
```

```
catch (SystemException se) {
   ...
}
```

This has exactly the same functionality as calling `string_to_object()`, except you do not have to call `narrow()` afterwards.

The method `string_to_object()` on `IE.Iona.OrbixWeb.CORBA.ORB` is overloaded to allow the individual fields of a stringified object reference to be specified. Refer to the section on `_OrbixWeb.ORB()` in the *OrbixWeb Programmer's Reference* for details on how to convert an instance of `org.omg.CORBA.ORB` to an instance of `IE.Iona.OrbixWeb.CORBA.ORB`.

The definition of this form of `string_to_object()` is as follows:

```
// Java
// In package IE.Iona.OrbixWeb.CORBA,
// in class ObjectRef.

public ObjectRef
   string_to_object(
        String host,
        String IFR_host,
        String ServerName,
        String marker,
        String IFR_server,
        String interfaceMarker);
```

The ability to create proxy objects from object reference strings has several useful applications. For example, this approach to proxy creation is often used in conjunction with the OrbixWeb Dynamic Invocation Interface (DII).

9

ORB Interoperability

ORB Interoperability allows communication between independently developed implementations of the CORBA standard. ORB interoperability enables a client of one ORB to invoke operations on an object in a different ORB via an agreed protocol. Thus, invocations between client and server objects are independent of whether they are on the same or different ORBs. The OMG has specified two standard protocols to allow ORB interoperability, GIOP and IIOP. This chapter discusses the use of these protocols.

The OMG-agreed protocol for ORB interoperability is called the General Inter-ORB Protocol (GIOP). GIOP defines the on-the-wire data representation and message formats. It assumes that the transport layer is connection-oriented. The GIOP specification aims to allow different ORB implementations to communicate without restricting ORB implementation flexibility.

The Internet Inter-ORB Protocol (IIOP) is an OMG defined specialisation of GIOP that uses TCP/IP as the transport layer. Specialised protocols for different transports (for example, OSI, Netware, IPX) or for new features, such as security, are expected to be defined by the OMG in due course.

There are many reasons why interoperability between the products of different ORB vendors is desirable. The core CORBA specification defines a standard for making invocations on an object via an ORB. A natural extension of this standard is that conforming implementations should allow invocations on objects from other conforming implementations. Within an organisation different ORBs may coexist reflecting separate

development effort or different ORB requirements by different parts of the organisation and at some point, these ORBs may need to communicate.

An overview of the GIOP and IIOP specifications is provided in this chapter. The example on page 218 shows how IIOP can be used in OrbixWeb.

Overview of GIOP

This section provides an overview of the elements of the GIOP specification. It is provided primarily as background information.

For full details of the GIOP specification, contact the OMG at the following Web site: `http://www.omg.org`.

Coding

The GIOP defines a transfer syntax known as Common Data Representation (CDR). CDR defines a coding for all IDL data types: basic types, structured types (including exceptions), object references and pseudo-objects such as `TypeCodes`.

All basic types are aligned on their natural boundaries. The architecture of the message sender determines whether the byte ordering is big-endian or little-endian. It is then the responsibility of the receiver to decode the message according to the byte ordering. Thus machines with common byte ordering may exchange messages without unnecessary byte swapping.

Message Formats

GIOP[1] defines eight message types. All messages include a common message header which includes the following information:

- The message size.
- A version number indicating the version of GIOP being used.
- The byte ordering.
- The message type.

1. These GIOP message formats are intended for internal use only.

Messages are exchanged between clients and servers. In this context, a client is an agent that opens connections and originates requests. A server is an agent that accepts connections and receives requests. The seven GIOP message types are as follows:

Request

A `Request` message is sent by a client to a server. It encodes an operation invocation which includes the identity of the target object, and an identifier used to match a `Reply` message to a `Request`. A `Request` may encode a get or set operation for an attribute.

Reply

A `Reply` message is sent by a server to a client. A `Reply` message encodes an operation invocation response, including `inout` and `out` parameters and exceptions.

A server receiving a `Request` message may not be able to provide direct access to the target object. This may be because the target object has moved or because the server receiving the `Request` message provides a location service. To indicate this, a `Reply` may contain a `LOCATION_FORWARD` status and an indication of the new location.

CancelRequest

A `CancelRequest` message may be sent from a client to a server to notify the server that a reply to a particular pending `Request` or `LocateRequest` message is no longer expected.

LocateRequest

A `LocateRequest` message may be used to probe for the location of a remote object. This might be appropriate where an operation's parameters are too large to transmit in a `Request` message that might return a `LOCATION_FORWARD` status. A `LocateRequest` message determines whether the target object reference is valid, whether the server can handle requests for that object or, if it returns a `LOCATION_FORWARD` status, indicates the location to which invocation on the reference should be sent.

LocateReply

A `LocateReply` message is sent by a server to a client in response to a `LocateRequest` message. It may contain a new IOR.

CloseConnection

A `CloseConnection` message is sent by a server to inform clients that it intends to close the connection. Any messages for which clients have not received a reply may be reissued on another connection.

MessageError

A `MessageError` message may be sent by a client or a server in response to any message whose message type or version number is unknown to the receiver of the message or whose message header is not properly formed.

The way in which these messages are used by an implementation of GIOP is transparent to the application. For example, a particular implementation may respond to a `LOCATE_FORWARD` status in a `Reply` message by transparently reissuing the call. Similarly, use of the `LocateRequest` message is an optional optimization.

Fragment

A `Fragment` message allows you to send a large message efficiently by transmitting the message as a sequence of fragments. Any `Request` or `Reply` message may be transmitted as fragments. The initial message is a `Request` or `Reply` message with a value in the GIOP header set to indicate that more fragments should be expected. The subsequent messages are then `Fragment` messages. `Fragment` messages are sent in the order in which they should be assembled.

Internet Inter-ORB Protocol (IIOP)

The mapping of GIOP message transfer to TCP/IP connections is called the Internet Inter-ORB Protocol (IIOP).

An object accessible via IIOP is identified by an *Interoperable Object Reference* (IOR). Since the format of normal object reference is not prescribed by the OMG, the format of an IOR includes an ORB's internal object reference as well as an internet host address and a port number. An IOR is managed internally by the interoperating ORBs. Refer to "Viewing Information about Object References" on page 228 for more details on IORs.

IIOP in OrbixWeb

OrbixWeb supports IIOP and the native Orbix protocol as alternative protocols. IIOP is the default protocol. Support for the Orbix protocol is provided primarily for backward compatibility.

You can indicate during compilation of an IDL definition which protocol should be used in the generated Java code for that definition. A client program can then make invocations on this definition and OrbixWeb automatically uses the chosen protocol. At this point, the chosen protocol is largely transparent at the application level.

Selection of Protocols

By default, code generated by the IDL compiler supports both IIOP and the Orbix protocol. When compiling IDL definitions, use the `-m` option with the following value to support the IIOP protocol only:

```
idl -m IIOPOnly
```

As described in Chapter 8 "Making Objects Available in OrbixWeb" on page 171, there are several ways in which a server can publish an object reference or IOR for retrieval by clients. IORs are required when using IIOP. OrbixWeb object references are required if using the Orbix Protocol. The protocol used does not affect the options available to application programmers.

Comparison of IIOP and the Orbix Protocol

IIOP has two important advantages over the Orbix protocol. The first is interoperability with other ORBs. The second is the availability of servers which have no platform-specific requirement, especially important in the Java domain.

Note: *All* servers which communicate using the Orbix protocol require an OrbixWeb daemon to run on the server host. This limits these servers to platforms where an OrbixWeb daemon is available. However, using IIOP, you can design client and server applications which have no external dependencies and are platform-independent.

For example, the following application pair would interoperate across ORBs, and also be platform-independent:

- A server which is not registered in the Implementation Repository, which creates and publishes IORs (for instance, using the Naming Service), and which calls the methods `ORB.connect()` and `ORB.disconnect()` instead of `impl_is_ready()` on the `ORB` object.
- A client which retrieves the IORs published by the server without calling the OrbixWeb `bind()` method.

Refer to Chapter 12, "Registration and Activation of Servers" on page 251 for details on how OrbixWeb servers can be run in a distributed system and their requirements in this context.

Example using IIOP in a Platform-Independent Application

This section illustrates the use of IIOP in OrbixWeb to create an interoperable application which does not rely on the availability of an OrbixWeb daemon process. The application developed here consists of a client and server as described in the example above. The server creates an IOR which it publishes using OrbixWeb Naming Service and then invokes `processEvents()` to handle client invocations on that IOR. The client retrieves the IOR using OrbixWeb Naming Service and invokes operations on the server object.

The example is based on the following IDL interface representing a two dimensional grid.

```
// IDL
interface grid {
   readonly attribute short height;
   readonly attribute short width;

   void set(in short row, in short col,in long value);
   long get(in short row, in short col);
};
```

Compiling the IDL Definition

The marshalling protocol uses IIOP by default. It is not necessary to specify the `-m` switch in order to use IIOP.

You can compile an IDL definition as normal:

```
idl -jP gridDemo grid.idl
```

Programming the Server

This section outlines the server code. It is assumed that an implementation of the Naming Service, such as OrbixWeb Naming Service is available and correctly installed. Following the convention used elsewhere in this guide, it is also assumed that class `gridImplementation` implements interface `grid`.

```
// Java
// Server main() method.

import CosNaming.*;

import org.omg.CORBA.SystemException;
import org.omg.CORBA.UserException;
import org.omg.CORBA.Object;

class gridserver {
   public static void main(String args[]) {
      // Assume TIE approach.
      grid gridImpl;
      ORB orb;

      // Declare Naming service types.
      Object initRef;
      NamingContext initContext;
      NamingContext objectsContext;
      NamingContext mathContext;
      NameComponent[] name;
```

```
try {
   // Create implementation object.
   gridImpl =
      new _tie_grid (new gridImplementation
                                    (100,100));
}
catch (SystemException se) {
   // Details omitted.
}

try {
   // Find initial naming context.
   orb = ORB.init(args,null);
   initRef =
      orb.resolve_initial_references
                          ("NameService");
   initContext = NamingContextHelper.narrow
                                     (initRef);

   // A CosNaming.Name is simply a sequence
   // of structs.
   name = new NameComponent[1];
   name[0] =
      new NameComponent("objects","");

   // (In one step) create a new context,
   // and bind it relative to the
   // initial context:
   objectsContext =
      initContext.bind_new_context (name);

   //reuse the NameComponent that has
   //already been created
   name[0].id = new String ("math");
   name[0].kind = new String ("");
```

```
            // (In one step) create a new context,
            // and bind it relative to the
            // objects context:
            mathContext =
               objectsContext.bind_new_context (name);

            name[0].id = new String ("grid");
            name[0].kind = new String ("");

            // Bind name to object gridImpl in context
            // objects.math:
            mathContext.bind (name, gridImpl);
         }
         catch (SystemException se) {
            // Details omitted.
         }
         catch (UserException ue) {
            // Use the exceptions defined in the
            // COSNaming IDL
         }
            // Call ORB.connect() to process
            // client invocations.
         orb.connect(gridImpl);
         try {
              Thread.sleep(1000*60*3);
         }
         catch (InterruptedException ex) {
            // Details omitted.
         }
      }
   }
```

This server instantiates a TIE object for interface `grid`. By default, OrbixWeb automatically identifies this object using an IOR. The server then resolves the initial context in the OrbixWeb Naming Service and associates the compound name `objects.math.grid` with the IOR, as described in Chapter 8 "Making Objects Available in OrbixWeb". Finally, the server enters an OrbixWeb event processing loop by calling `processEvents()`.

Programming the Client

This client program resolves the name `objects.math.grid` to locate the object reference published by the server using the Naming Service. The interoperable IOR retrieved from the Naming Service must be narrowed to an object reference of the appropriate interface before you can invoke operations in the normal way.

The source code for the client is as follows:

```
// Java
// Client application code.
// In file Client.java.

import CosNaming.*;
import IE.Iona.OrbixWeb._CORBA;

import org.omg.CORBA.SystemException;
import org.omg.CORBA.UserException;
import org.omg.CORBA.Object;

public class Client {
   public static void main (String args[]) {
      NamingContext initContext;
      NameComponent[] name;
      ORB orb;

      Object initRef, objRef;
      grid gRef;

      try {
         // Find initial naming context.
         orb = ORB.init(args,null);
         initRef =
            orb.resolve_initial_references
                              ("NameService");
         initContext = NamingContext.narrow
                                 (initRef);

         // Set up name and contexts.
         name = new NameComponent[3];
         name[0] =
            new NameComponent ("objects","");
```

```
            name[1] = new NameComponent ("math","");
            name[2] = new NameComponent ("grid","");


            // Resolve the name.
            objRef = initContext.resolve (name);
            gRef = grid.narrow (objRef);
         }
         catch (SystemException se) {
            // Details omitted.
         }
         catch (UserException ue) {
            // Use exceptions defined in the COSNaming
            // IDL
         }
         try {
            w = gRef.width();
            h = gRef.height();
         }
         catch (SystemException se) {
            // Details omitted.
         }

         System.out.println("height is " + h);
         System.out.println("width is " + w);

         try {
            gRef.set((short)2,(short)4,123);
            v = gRef.get((short)2,(short)4);
         }
         catch (SystemException se) {
            // Details omitted.
         }

         System.out.println(
            "value at grid position (2,4) is " + v);
      }
   }
```

Using IIOP and Binding from an OrbixWeb Client

It is possible to register an OrbixWeb IIOP server in the Implementation Repository, as illustrated throughout this guide. In this case, the OrbixWeb daemon becomes responsible for locating the server. It must also provide an IOR to the client on its initial invocation, and if necessary, activate the server. This has the drawback of introducing a dependency on an OrbixWeb daemon at the server host. However this approach does allow the server to take advantage of the OrbixWeb automatic launch facilities.

The mechanism required to register a server in the Implementation Repository is independent of the server protocol. However, an additional registration option is available to servers which use IIOP, as described in "Configuring an IIOP Port Number for an OrbixWeb Server" on page 226.

In order to use `bind()` to resolve IORs in an OrbixWeb client the following conditions must be satisfied:

- The server must be registered in the Implementation Repository, unless the OrbixWeb daemon is run with the `-u` switch

  Refer to "The OrbixWeb Java Daemon" on page 271 for more details.
- The server must call `impl_is_ready()` to initialize its server name.
- The client must be configured to use IIOP for OrbixWeb communications. This is the default case.

You can specify several configuration variables including the default protocol to be used when the client binds to a server object. In the context of using IIOP in a `bind()` call, a client would look as follows:

```
// Application client

import IE.Iona.OrbixWeb.CORBA.ORB;

public class javaclient1 {
      // Set bind communications protocol.
      ORB.setConfigItem("IT_BIND_USING_IIOP",
                        String.valueOf(true));

      // Port number for communicating with
      // OrbixWeb daemon using IIOP.
      ORB.setConfigItem("IT_ORBIXD_IIOP_PORT",1570);
```

```
        // Client performs bind ...
        // ...
    }
}
```

This default assignment for `IT_BIND_USING_IIOP` allows calls to `bind()` that use IIOP. You can change this to the following:

```
        // Set bind communications protocol.
        ORB.setConfigITem("IT_BIND_USING_IIOP",
                          String.valueOf(false));
```

In this case, calls to `bind()` can only use the Orbix protocol.

The `IT_ORBIXD_IIOP_PORT` setting specifies the port number on which the client tries to connect to the daemon using IIOP. You can also set this by configuring `OrbixWeb.properties` using the OrbixWeb Configuration Tool.

If you are using `orbixd`:

- The `IT_IIOP_PORT` variable in the daemon's `Orbix.cfg` file must be the same as the client's `IIOP_ORBIXD_IIOP_PORT` value.

If you are using `orbixdj` (the Java Daemon):

- The `IT_ORBIXD_IIOP_PORT` variable in the Java Daemon's `OrbixWeb.properties` file must be the same as the value of the client's `IT_ORBIXD_IIOP_PORT` variable.

Note: You should only use this code occasionally. Normally you should use the OrbixWeb Configuration Tool to set the default values. Refer to Chapter 4, "Getting Started with OrbixWeb Configuration" on page 53, for details on how to use the Configuration Tool.

Specifying a Communications Port for the OrbixWeb Daemon

It is important to note that the OrbixWeb daemon process listens for incoming OrbixWeb events on two communications channels:

- the OrbixWeb demon port (using Orbix protocol)
- the OrbixWeb daemon IIOP (using IIOP)

If a client wishes to invoke `bind()` using IIOP and `orbixd`, it should ensure that it communicates with the OrbixWeb daemon using the OrbixWeb daemon IIOP port and not the OrbixWeb daemon (Orbix protocol) port. The ports used by `orbixdj` support both IIOP and the Orbix Protocol, and are present purely for compatibility with `orbixd`. Refer to Chapter 4, "Getting Started with OrbixWeb Configuration" on page 53, and the *OrbixWeb Programmer's Reference* for a comprehensive discussion of configuration issues.

Configuring an IIOP Port Number for an OrbixWeb Server

Using IIOP, an OrbixWeb server must listen for client connection requests on a fixed TCP/IP port. The port number for each server is assigned by OrbixWeb on start-up.

In most cases this is done by the OrbixWeb daemon. Refer to the descriptions of `IT_DAEMON_SERVER_BASE` and `IT_DAEMON_SERVER_RANGE` in the *OrbixWeb Programmer's Reference* for more details.

When this approach is used, the port number assigned to a server subsequently becomes embedded in the contents of any IORs which that server creates. This approach has the drawback that a server which exits and is relaunched may no longer be able to recreate objects with IORs which exactly match those created in an earlier process. For this reason, OrbixWeb allows you to select a well-known IIOP port for each server program.

By default, the OrbixWeb daemon manages a well-known port for a server. This feature can be disabled by setting `IT_IIOP_USE_LOCATOR` to `false` in the server, as follows:

```
// Java
import IE.Iona.OrbixWeb.CORBA.ORB;
...

ORB.setConfigItem("IT_IIOP_USE_LOCATOR",""+ false);
```

This setting must be applied before any IORs are created in the server.

When registering a server in the Implementation Repository, you can specify a well-known port for a server using the `putit -port` switch, for example:

```
putit serverName -java -port portNumber ...
```

Note: The `-port` switch is supported by `orbixd` only.

If you set `IT_IIOP_USE_LOCATOR` to `true` and specify a port number for the server in this manner, the OrbixWeb daemon attempts to assign the required IIOP port to the server. If that port is not available and you are using `orbixd`, an attempt to create an IOR in the server raises a system exception.

If you set `IT_IIOP_USE_LOCATOR` to `true`, and do not specify a port number in a `putit` command, the OrbixWeb daemon assigns a default well-known port to the server.

A server which does not depend on the availability of an OrbixWeb daemon process should set `IT_IIOP_USE_LOCATOR` to false. In this case, an alternative mechanism is required to allow the server to establish a well-known IIOP port number. You can achieve this as follows:

```
// Server listen port for IIOP protocol.
ORB.setConfigItem("IT_IIOP_LISTEN_PORT",10,000);
```

This approach is only effective if the new value is assigned *before* the creation of any IORs in the server. The value of the `IT_IIOP_LISTEN_PORT` setting has no significance if `IT_IIOP_USE_LOCATOR` is set to `true`.

If you set `IT_IIOP_LISTEN_PORT` to zero, the server is not associated with a well-known port number. This means that an IIOP port is not dynamically assigned to the server on start-up.

Viewing Information about Object References

The **IOR Explorer** tool allows you to do the following:

- View information about object references.
- Import object references from a file.
- Save object references to a file.

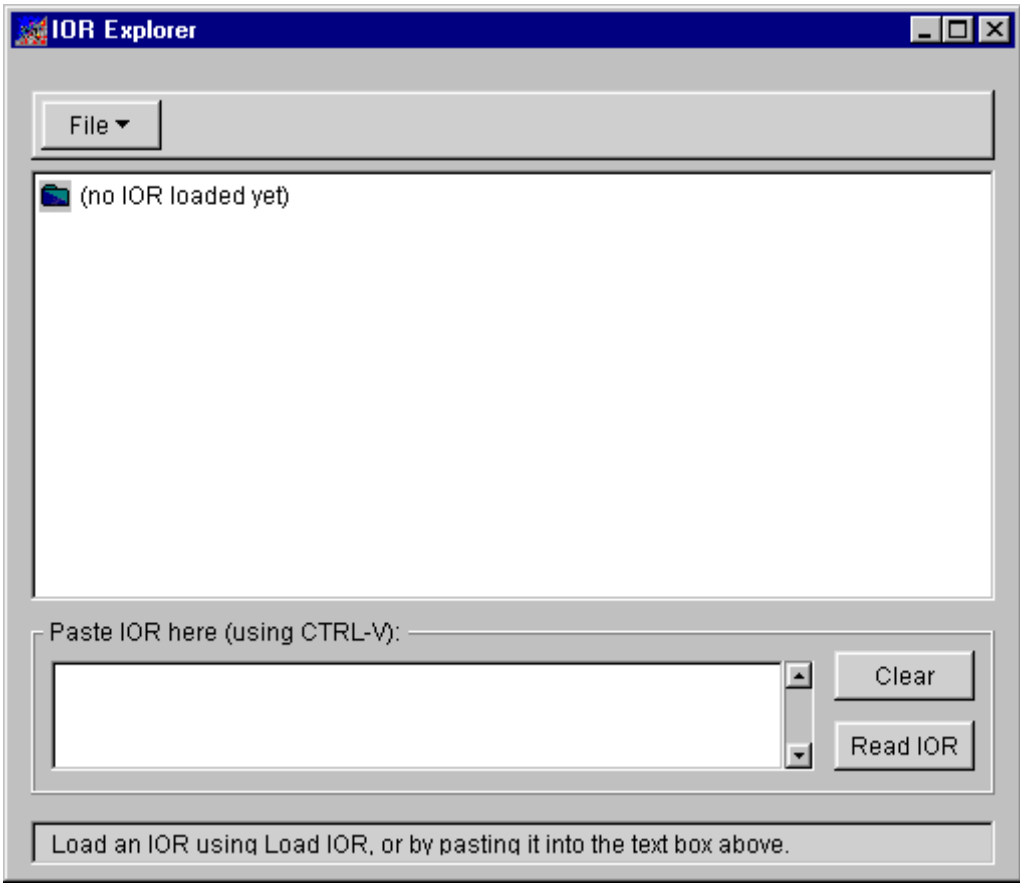

Figure 20: *The Interoperable Object Reference Explorer*

To start the object reference explorer, select the **IOR Explorer** option from the OrbixWeb main menu. The IOR Explorer appears as shown in Figure 20 on page 228. It consists of an IOR navigation area and an IOR entry area.

Importing an Object Reference into the IOR Explorer

If you bind a name to a CORBA object, you can import the reference for that object into the explorer as follows:

1. In the main browser window, navigate to the object that you wish to add to the explorer.
2. Select **Edit/Copy as**.
3. Select **Object Reference** in the resulting dialog box.
4. Highlight the IOR entry area in the IOR explorer and paste the copied IOR using **CTRL-V**.
5. Click on **Read IOR** to have it entered into the IOR navigation area.

Importing an Object Reference from a File

To import an object reference from a file, do the following:

1. Create a text file containing the object reference.

   You can obtain the string format of an object reference, for example, by calling the function `object_to_string()` on the object in your CORBA application.
2. In the explorer, select the **Load IOR** option in the **File** menu. The standard **Open File** dialog box for your operating system appears.
3. Enter the name of the file containing the object reference entry.
4. Click **OK**. The explorer displays the imported object reference in the navigation area.

Parsing an Object Reference

A object reference contains information about the location of a CORBA object. The explorer allows you to view the information contained in a CORBA Interoperable Object Reference (IOR). To view this information for an object reference added to the explorer, click on the "+" icons in the navigation area, as shown in Figure 21.

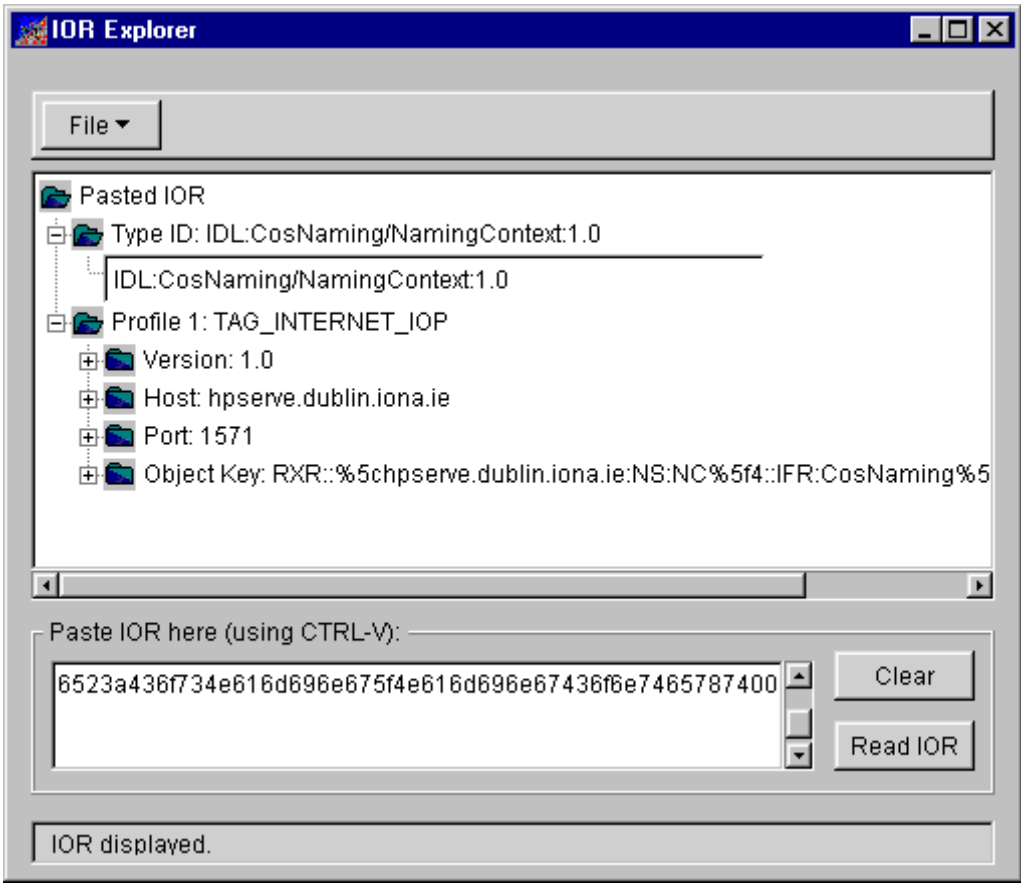

Figure 21: *The IOR Parse Dialog Box*

Interoperability between Orbix and OrbixWeb

The default protocol for the OrbixWeb runtime is IIOP. IIOP is also the default protocol for versions of Orbix 2.3 and above.

Earlier versions of Orbix use the Orbix protocol by default. If you are using code generated by older versions of Orbix, you must select one protocol. If you choose IIOP, the C++ server must be linked with the IIOP library. An example of this is provided in the `GRID_IIOP` demonstration supplied with Orbix.

If you choose the Orbix protocol, the Java client must include the line:

```
ORB.setConfigItem("IT_BIND_USING_IIOP",""+false);
```

Part III

Running OrbixWeb Programs

10

Running OrbixWeb Clients

This chapter deals with running OrbixWeb client applications and applets, and provides information on some general runtime issues for clients.

Running Client Applications

The procedure for running an OrbixWeb client application is similar to the procedure for running any stand-alone Java application. In general, you must fulfil three requirements:

- Obtain access to the Java bytecode for the application.
- Make this code available to the Java bytecode interpreter.
- Run the interpreter on the class which contains the `main()` method for the application.

The only runtime difference between an OrbixWeb application and a standard Java application lies in the first of these requirements. An OrbixWeb application must be able to access the classes stored in the `IE.Iona.OrbixWeb` and `org.omg.CORBA` packages. It also requires access to the classes produced by compiling the IDL definitions referenced by the application. The `IE.Iona.OrbixWeb` and `org.omg.CORBA` packages are usually installed in the OrbixWeb `classes` directory. The `org.omg.CORBA` classes are portable and may already be installed in the runtime environment.

How you make class location information available to the Java interpreter is dependent on the Java development environment you use. However, you should indicate the location of the following:

- The OrbixWeb packages.
- The Java API classes.
- The IDL compiler output classes.
- The application-specific classes.

For example, if you are using the `java` interpreter from Sun Microsystems JDK, you should add the location of each to the `CLASSPATH` environment variable or specify this information in the `-classpath` switch.

OrbixWeb offers a set of convenience tools, called wrapper utilities. These make information about defaults automatically available to the Java interpreter and the Java compiler. The wrapper utilities, `owjava` and `owjavac`, are described in the section "Using the Wrapper Utilities" on page 240.

Similarly, how you run the application through the interpreter may differ between development environments. Again, if you are using the JDK `java` interpreter, you can pass the name of the class that contains the application `main()` method to the interpreter command, as follows:

```
java <class name>
```

Running OrbixWeb Client Applets

The requirements for running an OrbixWeb client applet are slightly more complex than those for an application. To display a Java applet, you should reference the applet class in a HTML file using the HTML `<APPLET>` tag, and then load this file into an applet viewer or a Java-enabled web browser. The runtime requirements for the applet depend on whether it is loaded directly from a HTML file or downloaded from a web server.

Loading a Client Applet from a File

When you load an OrbixWeb client applet from a file, the runtime requirements are similar to those for running a client application. You should do the following:

- Obtain access to the Java bytecode for the applet.
- Make this code available to the Java bytecode interpreter embedded in the browser.
- Load the HTML file that references the applet into the browser.

The second of these requirements often translates to setting the `CLASSPATH` environment variable appropriately *before* running the viewer or browser and loading the applet. This variable should usually include the location of the following:

- The OrbixWeb package classes.
- The Java API classes.
- The IDL compiler output classes.
- The other applet-specific classes.

If you use a Java-enabled browser, the location of the Java API classes is generally not required. In some cases, the location of the `org.omg.CORBA` package is also not required.

When loading an OrbixWeb client applet from a file, you can specify a `codebase` attribute in the HTML `<APPLET>` tag to specify the location of the required class files. The next section describes how you can do this.

Note: When loading an OrbixWeb applet from a file, you should use a recent browser version. There are some browser-based URL restrictions associated with early browser versions.

Loading a Client Applet from a Web Server

If an OrbixWeb applet is loaded into a browser from a Web server, you cannot specify access paths for the required Java classes at runtime. In this case, you should provide access to all the classes the applet requires in a single directory. Then, instead of setting an

environment variable, you can use the `codebase` attribute of the HTML tag `<APPLET>` to indicate the location of the applet bytecode.

For example:

```
<APPLET codebase=<applet class directory>
   code=<applet class file>
```

If you use a Java-enabled Web browser to view an applet, you do not need to provide access to the Java API classes, because these are already available.

Security Issues for Client Applets

The necessity of strict security restrictions in Java applets is well documented. There are two primary security restrictions on applets:

- No access to local file systems.
- Limited network access.

Both of these restrictions are imposed by the browser sandbox, and apply to all applets, regardless of how they are loaded.

Applets do not have access to the file system of the host on which they execute. They cannot save files to the system or read files from it. Any OrbixWeb client implemented as a Java applet must obey this restriction.

In order to prevent the violation of system integrity, Web browsers often limit the network connectivity of applets which are downloaded from a Web server. Such applets can only communicate with the host from which they were downloaded.

This limitation has obvious implications for OrbixWeb client applets downloaded from Web servers. In particular, such clients can only communicate directly with OrbixWeb servers located on the host from which the clients themselves were downloaded. If this restriction applies to an OrbixWeb client applet, attempts by that client to bind to a server on an inaccessible host raises a system exception of type `org.omg.CORBA.COMM_FAILURE`.

Note: Using Wonderwall allows OrbixWeb client applets to be granted access to servers on hosts other than those from which they were downloaded. Refer to the *Wonderwall Administrator's Guide* for more details.

The exact details of applet security are dependent on the browser implementation and may exceed the restrictions described here. Newer browsers allow security to be configured for signed applets. Consult your browser documentation for further information.

Debugging OrbixWeb Clients

An OrbixWeb client application or applet has the same fundamental characteristics as any other Java program. You can debug OrbixWeb clients with any available Java debugging tool, for example, the JDK `jdb` debugger.

When debugging OrbixWeb clients, it is especially important to be aware of Java exceptions thrown during OrbixWeb method invocations. OrbixWeb provides a set of system exceptions indicating various categories of execution errors. These represent vital information for locating the source of invocation failures in a distributed application. You can handle these exceptions in client code by using Java `try` and `catch` statements. Similarly, they can be handled like standard Java exceptions when using a Java debugger.

For more details on OrbixWeb integration with Java exceptions, refer to Chapter 15 "Exception Handling" on page 295.

Possible Platform Dependencies in OrbixWeb Clients

In general, OrbixWeb clients are only dependent on the availability of a Java interpreter on the target execution platform. However, you should be aware of two issues that may affect the platform-independence of an OrbixWeb system:

- Using locators.
- Using the `bind()` method.

Using Locators

First, if a client depends on the OrbixWeb locator mechanism to find a target server, as described in Chapter 25 "Locating Servers at Runtime" on page 473, the client requires an OrbixWeb daemon to run on its local host. Otherwise, the server location mechanism fail. This limits such a client to running on platforms where an OrbixWeb daemon is available. In the case of OrbixWeb client applets, an OrbixWeb daemon must be running on the machine from which they were downloaded.

Using bind()

Second, if a client uses the OrbixWeb `bind()` method to create a proxy for a server object, the `bind()` call fails unless an OrbixWeb daemon is available at the server host. Consequently, a client using `bind()` does not execute successfully unless the target server is restricted to running on a host where an OrbixWeb daemon is available.

Using the Wrapper Utilities

The OrbixWeb Wrapper Utilities, `owjava` and `owjavac`, are convenience tools designed to act as a front end to the Java interpreter and Java compiler, respectively. This section outlines the use of these tools, and also describes the standard Java command line equivalent.

Consider the following standard command line entry to invoke the Java interpreter:

```
C:\JDK\bin\java -classpath C:\JDK\lib\classes.zip;C:\OW31\classes
-DOrbixWeb.config=C:\OW31\classes\OrbixWeb.properties
myPackage.myClass
```

Using the `owjava` wrapper utility, you can reduce the standard command line entry to the following:

```
owjava myPackage.myClass
```

You must access both `owjava` and `owjavac` from the command line. Both of these have an equivalent function on UNIX and Windows. The examples shown in this chapter apply to both Unix and Windows, apart from obvious differences in paths.

Using owjava as a Front End to the Java Interpreter

The `owjava` wrapper is a front end for the Java interpreter you are using, designed for use with OrbixWeb. It takes all the same arguments as your chosen Java interpreter and passes them on, together with some other defaults.

`owjava` uses the `ORBIXWEB_HOME` registry/environment variable to find the `OrbixWeb.properties` file. From there it reads the full path of the Java interpreter, the default classpath and the name of the switch the Java interpreter uses to specify its class path. For example, Microsoft J++ uses `-c` whereas all other Java Development Kits use `-classpath`.

By default, `owjava` passes the default classpath and a variable containing the path of the `Orbixweb.properties` file to the Java interpreter.

So, for example, if OrbixWeb is installed in `C:\OW31` and the JDK is installed in `C:\JDK`, calling `owjava` as follows:

```
owjava myPackage.myClass
```

executes the following command:

```
C:\JDK\bin\java -classpath C:\JDK\lib\classes.zip; C:\OW31\classes
-DOrbixWeb.config=C:\OW31\classes\OrbixWeb.properties
myPackage.myClass
```

You can override this standard behaviour by using the OrbixWeb Configuration Tool to change the settings. Refer to Chapter 4, "Getting Started with OrbixWeb Configuration" on page 53, for details on the Configuration Tool.

Using owjavac as a Front End to the Java Compiler

This tool acts as a front end to your chosen Java compiler, and is designed for use with OrbixWeb. Its behaviour is similar to the `owjava` tool described previously, but the defaults are different. By default, `owjavac` passes the default `CLASSPATH` and the `CLASSES` directory to the compiler.

So, for example, if OrbixWeb is installed in `C:\OW31` and the JDK is installed in `C:\JDK`, calling `owjavac` as follows:

```
owjavac -d C:\OW31\classes\myClass.java
```

executes the following command:

```
C:\JDK\bin\javac -classpath C:\JDK\lib\classes.zip;C:\OW31\classes
-d C:\OW31\classes\ myClass.java
```

You can override this standard behaviour by using the OrbixWeb Configuration Tool to change the relevant settings. Refer to Chapter 4, "Getting Started with OrbixWeb Configuration" on page 53, for more details on the Configuration Tool.

Using the Interpreter and Compiler without the Wrapper Utilities

You do not need to use the Wrapper Utilities. These are provided as convenience tools only. You can use the standard Java command line format for `java` and `javac`, by using the formats specified as follows:

Using the java Command

```
C:\JDK\bin\java -classpath C:\JDK\lib\classes.zip; C:\OW31\classes
-DOrbixWeb.config=C:\OW31\classes\OrbixWeb.properties
myPackage.myClass
```

Using the javac Command

```
C:\JDK\bin\javac -classpath C:\JDK\lib\classes.zip;
C:\OW31\classes -d C:\OW31\classes\ myClass.java
```

11

Using OrbixWeb on the Internet

OrbixWeb client applets are, like any applet, subject to security restrictions imposed by the browser in which they execute. The most fundamental of these restrictions include the inability to access local disks and the inability to contact an arbitrary internet host. This chapter describes techniques that allow client applets to get around these restrictions in a secure manner. The first technique involves IONA's WonderWall which is a full IIOP firewall proxy. The second technique involves the use of signed applets.

About Wonderwall

OrbixWeb provides inbuilt support for the Wonderwall. You can use Wonderwall in two main ways:

- As a full *firewall proxy* which can filter, control and log your IIOP traffic.
- As a simple *intranet request-routing server* which passes IIOP messages from your applet, via the Web server, to the target server.

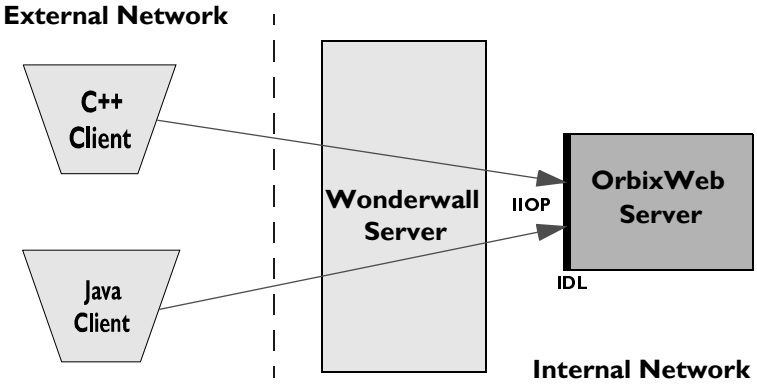

Figure 22: *Using OrbixWeb and Wonderwall*

Using the Wonderwall with OrbixWeb as a Firewall Proxy

To run the Wonderwall in a traditional secure mode, use the file `secure.cf`. The Wonderwall command is as follows:

```
iiopproxy -config secure.cf
```

This mode of operation requires that the target objects and operations be listed in the configuration file. For further details, refer to the *Wonderwall Administrator's Guide*. This provides a guide to using Wonderwall's access control lists and object specifiers.

OrbixWeb Configuration Parameters Used to Support the Wonderwall

OrbixWeb has automatic inbuilt support for the Wonderwall. This means that if a connection attempt fails using the default direct socket connection mechanism, OrbixWeb can transparently attempt to connect to any IIOP servers via the Wonderwall. This also means that Wonderwall can be used to:

- Provide HTTP Tunnelling for OrbixWeb-powered Java applets and applications.
- Provide automatic intranet routing capability for OrbixWeb-powered Java applets, to avoid browser security restrictions.
- Use OrbixWeb applications and applets with the Wonderwall, with no code changes.

Configuring OrbixWeb to Use the Wonderwall

To use the Wonderwall with OrbixWeb, you must supply OrbixWeb with the location of the Wonderwall. You should use the following configuration parameters:

- `OrbixWeb.IT_IIOP_PROXY_HOST`

  This contains the name of the host on which the Wonderwall is running.
- `OrbixWeb.IT_IIOP_PROXY_PORT`

  This contains the IIOP port on which the Wonderwall is running.

You can set these configuration parameters using any of the following:

- The OrbixWeb Configuration Tool.
- The `Config.setConfigItem()` call.
- Other OrbixWeb configuration mechanisms, such as applet tags, system properties or command-line options.

For example, the following is a fragment of a HTML file which uses applet-tag parameters:

```
<applet code=GridApplet.class height=300 width=400>
   <paramname="OrbixWeb.IT_IIOP_PROXY_HOST"value="wwall.iona.com">
   <param name="OrbixWeb.IT_IIOP_PROXY_PORT" value="1570">
</applet>
```

Configuring OrbixWeb to Use HTTP Tunnelling

HTTP Tunnelling is a mechanism for traversing client-side firewalls. Each IIOP Request message is encoded in HTTP base-64 encoding, and a HTTP form query is sent to the Wonderwall, containing the IIOP message as query data. The IIOP Reply is then sent as a HTTP response.

Using HTTP Tunnelling allows your applets to be used behind a client's firewall, even when a direct connection, or even a DNS lookup of the Wonderwall hostname, is impossible.

To use HTTP Tunnelling, you must use the `ORB.init()` API call to initialize OrbixWeb. The call to initialize OrbixWeb from inside an applet's `init()` method is as follows:

```
public void init () {
      // Initialize the ORB.
      IE.Iona.OrbixWeb.CORBA.ORB.init (this, null);

// Your applet initialization code can continue below
         ...
}
```

This allows OrbixWeb to retrieve the `codebase` from which the applet was loaded. The `codebase` is then used to find the Wonderwall's interface for HTTP Tunnelling, a pseudo-CGI-script called "`/cgi-bin/tunnel`". For more information on use of the `codebase` in Java, see the Javasoft Web site, at `http://www.javasoft.com/`.

The Wonderwall should be used as the Web server that provides the applet's classes, because an untrusted Java applet is only permitted to connect to the Web server named in the codebase parameter. However, you can provide the HTML and images for your main Web site from another Web server, such as Apache, IIS or Netscape, and simply refer to the Wonderwall Web server in the applet tag, as follows:

```
//For example, in file http://www.iona.com/demo.html

<applet code=GridApplet.class
   codebase=http://wwall.iona.com/GridApplet/classes
   height=300 width=400>
</applet>
```

With this setup, your HTML and images are loaded from the main Web site (`www.iona.com`), yet your applet code is loaded from `wwall.iona.com`. As a result the applet is permitted to open connections to that host. For greater efficiency, you should make a `ZIP`, `JAR` and/or `CAB` file containing the classes used by your applet, and store these on the main Web site also. The Web browser downloads these from the main site, and

does not need to load the classes from the Wonderwall site. This is a generally recommended practice, even if you are not using Wonderwall.

You can also provide a Wonderwall set-up to support HTTP Tunnelling on the same machine as the real HTTP server. This requires that the Wonderwall runs on a different port from the main server. Some sites may only allow outgoing HTTP traffic on port `80`, the standard port, so this could restrict the potential audience for your applet slightly.

You should ensure that the applet's classes are available in the directory you named in the `codebase` URL. In the example above, this would be `GridApplet/classes`. This directory path is relative to the directory named in the `http-files` parameter of your Wonderwall configuration file.

If you wish an application to use HTTP Tunnelling, or would prefer to override an applet's HTTP Tunnelling setup, the following three configuration parameters are provided:

- `OrbixWeb.IT_HTTP_TUNNEL_HOST`

  This contains the name of the host on which the Wonderwall is running.

- `OrbixWeb.IT_HTTP_TUNNEL_PORT`

  This contains the HTTP port on which the Wonderwall is running.

- `OrbixWeb.IT_HTTP_TUNNEL_PROTO`

  This contains the protocol used. Currently the only protocol value supported for HTTP Tunnelling is "`http`". Refer to "Configuring OrbixWeb to Use the Wonderwall" on page 245 for more details on how to set these parameters,

The Wonderwall supports HTTP 1.1 and HTTP 1.0's Keep-Alive extension. This means that more than one HTTP request can be sent across TCP connections between the client and the Wonderwall (or between a HTTP proxy and the Wonderwall). This greatly increases the efficiency of HTTP.

Manually Configuring OrbixWeb to Test Tunnelling

In order to test HTTP Tunnelling or IIOP via the Wonderwall, OrbixWeb provides two more configuration parameters:

- `OrbixWeb.IT_IIOP_PROXY_PREFERRED`
- `OrbixWeb.IT_HTTP_TUNNEL_PREFERRED`

If you set either of these parameters to `true`, the relevant connection mechanism is tried first, before the direct connection is attempted. IIOP Proxying takes precedence over HTTP Tunnelling, so if you enable both of these parameters, IIOP Proxying is tried.

Using the Wonderwall as an Intranet Request-Router

The Wonderwall can also be used as an intranet request router for IIOP, providing a means by which your OrbixWeb applets can contact servers that reside on hosts other than the host on which your Web server is running. The file `intranet.cf` is used in this configuration, so the Wonderwall command is as follows:

```
iiopproxy -config intranet.cf
```

Refer to the *Wonderwall Administrator's Guide* more details on using the Wonderwall as an intranet request router.

This mode of operation requires no configuration. Using the Wonderwall, any server can be connected to, and any operation can be called.

Applet Signing Technology

For security reasons, an applet is prevented from accessing the local file system and connecting to a host other than the host from which it was downloaded. Often these restrictions must be relaxed, in order for an applet to be fully functional. It is possible to achieve this using signed applet technology.

A signed applet has a digital signature which is interpreted as a sign of good intent. An applet that has been signed with a trusted digital signature may therefore be treated more permissively by a browser, and may even be granted the permission of a full application.

The following section provides a brief overview of signed applet technology. More detailed information is available on-line in the IONA Knowledge Base. See the IONA Web site at: `http://www.iona.com/`

Overview

There is no single standard implementation of applet-signing technology, however the implementations offered by Netscape and Microsoft are widely adopted. Specific details of these vendors implementations are available from their corporate Web sites. In this section, discussion is limited to the implementation independent characteristics of the technology.

How Applets are Signed

Applets may be signed using public key cryptography technology. Distributors of the applet must digitally sign the applet with their private key. When an applet thus signed is downloaded by a browser, it can determine the identity of the signing entity by consulting a Certification Authority. A Certification Authority is a trusted third party that verifies the identify of a key holder. The browser may also determine whether the applet has been tampered with. Assuming there are no problems, the browser may assume that the applet is not malicious, and grant it extended privileges.

The user must ultimately grant the applet these extended privileges, either by configuring browser security settings or responding at runtime to individual requests for privileges from the applet. In some circumstances it may be the case that an applet does not function correctly unless it is granted extended privileges.

The benefits of signed applet technology to the OrbixWeb applet programmer include the following:

- The ability to contact any host.
- The ability to cache information locally on disk.
- The ability to access system properties.

It is common for the applet, other classes it requires and associated files to be bundled into a single `archive` file. In this case, it is the archive that is signed and downloaded to the browser, thereby reducing download time.

Looking Ahead

It is expected that browsers will be able to support multiple archives in the future. Deployment should then become more flexible and efficient since applications can be split into a number of archives, each containing classes pertaining to a particular area of functionality. For example, an OrbixWeb applet may be split into archives containing the OrbixWeb runtime, the Java classes generated by the IDL compiler, the applet code and finally third party archives.

The OrbixWeb installation includes Microsoft CAB (signed) and Netscape JAR (unsigned) compatible archives. They can be found in the `classes` directory of your OrbixWeb installation.

12

Registration and Activation of Servers

This chapter describes the Implementation Repository, which is effectively a database of server information. This is the component of OrbixWeb that maintains registration information about servers and controls their activation. The Implementation Repository is implemented in the OrbixWeb daemon. The OrbixWeb daemon and utilities provide a superset of the functionality supported by a standard, non-Java Orbix installation.

This chapter outlines the full functionality supported by the OrbixWeb Implementation Repository. It also discusses aspects of registration and activation that affect servers communicating over the CORBA Internet Inter-ORB Protocol (IIOP) or the Orbix protocol. Aspects of server activation that are specific to IIOP servers are also described. IIOP servers only need to be registered in the Implementation Repository under certain circumstances, and this can be advantageous in a Java environment.

The Implementation Repository

The OrbixWeb Implementation Repository maintains a mapping from a server's name to the Java program which implements that server. A server must be registered with the Implementation Repository to make use of this mapping.

If the server is not running, it is launched automatically by OrbixWeb when a client *binds* to one of the server's objects, or when a client *invokes an operation* on an object reference which names that server. The OrbixWeb daemon launches a Java server by invoking the Java interpreter on the class specified in an Implementation Repository entry.

To allow the daemon to correctly locate and invoke the Java interpreter, it is important that the values `IT_JAVA_INTERPRETER` and `IT_DEFAULT_CLASSPATH` are correctly configured. The configuration of these values is described in the chapter "OrbixWeb Configuration" in the *OrbixWeb Programmer's Reference*.

When a client first communicates with an object, OrbixWeb uses the Implementation Repository to identify an appropriate server to handle the connection. This search can occur in the following circumstances:

- During a call to `bind()`, if pinging is enabled, otherwise, on the first invocation on an object reference returned by `bind()`.

  You can call the method `ORB.pingDuringBind()` (in package `IE.Iona.OrbixWeb.CORBA`) on the `_CORBA.Orbix` object to configure this. If this is set to `true`, pinging is enabled. If this is `false`, the server is not launched automatically when a bind occurs.
- During a call to the method `ORB.string_to_object()`.
- When an object is used for the first time after being received as a parameter or return value via an intermediate server.

If a suitable entry cannot be found in the Implementation Repository during a search for a server, a system exception is returned to the caller.

Activation Modes

OrbixWeb provides a number of different mechanisms, or *modes*, for launching servers, giving you control over how servers are implemented as processes by the underlying operating system. The mode of a server is specified when it is being registered.

Note: The availability of a given activation mode depends on which OrbixWeb daemon (`orbixd` or `orbixdj`) is used. The default activation modes are available to both `orbixd` and `orbixdj`, and are sufficient for most applications. Refer to "The OrbixWeb Java Daemon" on page 271 for further information on `orbixdj`.

Primary Activation Modes

The following primary activation modes are supported.

Shared Activation Mode (Default)

This mode is supported by `orbixd` and `orbixdj`.

In this mode, all of the objects with the same server name on a given machine are managed by the *same* process on that machine. This is the most commonly-used activation mode.

If the process is already launched when an operation invocation arrives for one of its objects, OrbixWeb routes the invocation to that process. Otherwise OrbixWeb launches the process, using the Implementation Repository's mapping from server name to class name and class path.

Unshared Activation Mode

This mode is supported by `orbixd` only.

In this mode, individual objects of a server are registered with the Implementation Repository. All invocations for an individual object are handled by a single process. This server process is activated by the first invocation of that object. Thus, one process is created per active registered object. Each object managed by a server can be registered with a different Java class, or any number of them can share the same class.

per-method Call Activation Mode

This mode is supported by `orbixd` only.

In this mode, individual operation names are registered with the Implementation Repository. You can make inter-process calls can be made to these operations, and each invocation results in the creation of an individual process. A process is created to handle each individual operation call, and the process is destroyed once the operation has completed. You can specify a different Java class for each operation, or any number of them can share the same class.

Secondary Activation Modes

For each primary activation mode, a server can also be launched in one of the following secondary activation modes.

multiple-client (Default)

This mode is supported by `orbixd` and `orbixdj`.

In this mode, activations of the same server by different users or *principals* will share the same process, in accordance with whichever fundamental activation mode is selected.

per-client

This mode is supported by `orbixd` only.

In this mode, activations of the same server by different end-users will cause a different process to be created for each such end-user.

per-client-process

This mode is supported by `orbixd` only.

In this mode, activations of the same server by different client processes causes a different process to be created for each such client process.

Persistent Server Mode

If a server is registered in the shared mode, it can be launched manually prior to any invocations on its objects. Subsequent invocations are passed to the process. CORBA uses the term *persistent server* to refer to a process launched manually in this way. The OMG CORBA term "persistent server" is not ideal, because it can be confused with the notion of persistent (long lived, on disk) objects. It may be more useful to view a "persistent" server as a manually launched server.

Launching persistent servers is useful for a number of reasons. Some servers take considerable time to initialize, and therefore it makes sense to launch these servers before clients wish to use them. Also, during development, it may be clearer to launch a server in its own window, allowing its diagnostic messages to be more easily seen. You can launch a server in a debugger during the development stage to allow debugging.

Since OrbixWeb uses the standard OMG IDL-to-Java mapping, all clients and servers must call `org.omg.CORBA.ORB.init()` to initialize the ORB. A reference to the `ORB` object is returned. You can invoke the ORB methods defined by the standard on this instance. Refer to the description of `org.omg.CORBA.ORB` in the *OrbixWeb Programmer's Reference* for more details on this topic.

Manually launched servers, once they have called `impl_is_ready()`, behave in a similar way to shared activation mode servers. If a server is registered as unshared or per-method, `impl_is_ready()` fails if the server is launched manually. Refer to "Persistent Servers" on page 261 for more details.

Note: If you are using `orbixd`, a shared server may be registered so that it may *only* be launched manually. This means that OrbixWeb does not launch the server when an operation invocation arrives for one of its objects. This is explained in "Unregistered Servers" on page 262. Use the `-persistent` with `putit` to register a server so that it may only be launched manually.

Usually, clients are not concerned with the activation details of a server or aware of what server processes are launched. To a client, an object in a server is viewed as a stand alone unit; an object in a server can be bound to and communicated with without considering activation mode details.

Although servers are registered in the Implementation Repository, you do not need to register individual objects; only those objects for which OrbixWeb should launch a process.

Implementation Repository Entries

An entry for a server in an Implementation Repository includes the following information:

- The server name.
  Server names may be hierarchical, so the Implementation Repository supports nested directories.
- The primary activation mode (shared, unshared, or per-method).
- The secondary activation mode (per-client, per-client-process or multiple-client).
- Whether the server is a persistent-only server—it can only be launched manually.
- The server owner—the user who registered the server.
- Permissions specifying which users have the right to launch the server and which users have the right to invoke operations on objects in the server.
- A set of *activation orders* specifying a marker or method and a launch command for that marker or method. For the shared or unshared activation modes, a number of activation orders may exist for different markers. For the per-method activation mode, a number of activation orders may exist for different methods.

putit

The `putit` command creates an Implementation Repository entry, if no entry exists, for the specified server. If an Implementation Repository entry already exists for the server, the `putit` command creates or modifies an activation order within the existing entry. In the latter case, the `putit` command must specify the *same* fundamental activation mode (shared, unshared or per-method) as that already registered for the server.

catit

The `catit` command displays the information on a server in an Implementation Repository entry. Alternatively, the you can use the Server Manager tool. Refer to the *OrbixWeb Programmer's Reference* for details of how to use this tool.

The OrbixWeb putit Utility for Server Registration

The `putit` utility registers servers with the Implementation Repository. This section outlines some examples of common uses of `putit`. A full description of `putit` and its switches is given in the *OrbixWeb Programmer's Reference*.

The `putit` command is used most often in either of the following forms:

```
putit serverName -java
   -classpath <full classPath> className

putit serverName -java
   -addpath <partial ClassPath> className
```

The first command form indicates that the server is to be registered with the specified complete class path, independent of any configuration settings, with the specified class name.

The second command form indicates that the specified class path should be appended to the value of `IT_DEFAULT_CLASSPATH` in the `OrbixWeb.properties` file, when the daemon attempts to launch the server.

The `-java` switch is an extension of the standard Orbix `putit` command This indicates that the specified server should be launched by the Java interpreter. You can truncate this switch to `-j`.

By default, `putit` uses the *shared* activation mode. Therefore, on any given host, all objects with the specified server name are controlled by the same process. Also by default, `putit`

registers a server in the *multiple-client* activation mode. This means that all client processes bind to the same server process. For example:

```
putit Bank -java -addpath
   /usr/users/chris/banker bank_demo.BankServer
```

In this example, the class `bank_demo.BankServer` is registered as the implementation code of the server called `BankSrv` at the current host. A partial class path of `/usr/users/chris/banker` is also specified. The `putit` command does not launch the server. You can do this explicitly from the shell or otherwise. Alternatively, OrbixWeb may automatically launch the server in shared mode in response to an incoming operation invocation.

Server names may be hierarchically structured, in the same way as UNIX file names. For example:

```
putit banks/BankSrv -java -addpath
   /usr/users/chris/banker bank_demo.BankServer
```

Hierarchical server names are useful in structuring the name spaces of servers in Implementation Repositories. You can create the hierarchical structure using the `mkdirit` command. Alternatively, you can use the OrbixWeb Server Manager tool. Refer to the *OrbixWeb Programmer's Reference* for details on both of these methods.

Examples of Using putit

The following examples illustrate some further switches to `putit`.

-unshared

If you are using the `orbixd` as your daemon process, you can use the `-unshared` switch to register a server in the unshared activation mode:

```
putit -unshared NationalTrust -java -classpath
   /classes:/jdk/classes:/tmp/bank bankPackage.BankServer
```

This command registers an unshared server called "`NationalTrust`" on the local host, with the class name and full class path. Each activation for an object goes to a unique server process for that particular object. All users accessing a particular object share the same server process.

-marker

You can specify a marker to the `putit` command to identify an object to which `putit` applies:

```
putit -h alpha -marker Boston NationalBank -java -addpath
   /bank/classes:/local/classes bankPackage.BankServer
```

This command registers a shared server called "`NationalBank`", with the specified class name and partial class path. However activation only occurs for the object whose marker matches "`Boston`". There is at most one server process resulting from this registration request. Other `-marker` registrations can be issued for server `NationalBank` for other objects in the server. All users accessing the "`Boston`" object share the same server process.

The `-h` switch specifies the host name on which to execute the `putit` command.

Additional Registration Commands

Implementation Repository entries created by `putit` can be managed using the following commands:

`catit`	Outputs full details of a given Implementation Repository entry.
`chmodit`	Allows launch and invoke rights on a server to be granted to users other than the server owner.
`chownit`	Allows the ownership of Implementation Repository entries and directories to be changed.
`killit`	Kills a running server process.
`lsit`	Lists a specific entry or all entries.
`mkdirit`	Creates a new registration directory. You can structure the Implementation Repository hierarchically like UNIX file names.
`pingit`	Pings the OrbixWeb daemon to determine whether it is alive.
`psit`	Outputs a list of server processes known to the OrbixWeb daemon.
`rmdirit`	Removes a registration directory.

`rmit`	Removes an Implementation Repository entry or modifies an entry.

Execute any of these commands without arguments to obtain a summary of its switches.

Further Mode Options: Activation and Pattern Matching

Recall from Chapter 8, "Making Objects Available in OrbixWeb" , that a server programmer may choose the marker names for objects. Alternatively, they can be assigned automatically by OrbixWeb.

Pattern Matching using orbixd

Pattern matching functionality for markers is supported by `orbixd` only. Because objects can be named, the various activation policies can be instructed to use pattern matching when seeking to identify which server process to communicate with. In particular, when a server is registered, you can specify that it should be launched if any of a set of its objects are invoked. You can specify this set of objects by registering a marker pattern which uses wild card characters. If no pattern is specified, invoking on any of a server's objects causes the server to be launched, if it has not already been launched.

You can also specify patterns for methods so that operation names matching a particular pattern cause a particular server to be launched.

Pattern matching functionality for markers is not currently supported by `orbixdj`.

Persistent Servers

Persistent servers refer to those that are launched manually. You should ensure that the persistent server name is correctly set *before* it has any interaction with OrbixWeb. For example, a persistent server should not pass out an object reference for one of its objects (as a parameter or return value, or even by printing its object reference string) until the server name has been set.

The following methods provide two approaches in OrbixWeb to launching servers manually:

- `_CORBA.Orbix.impl_is_ready()`
- `ORB.connect()`

_CORBA.Orbix.impl_is_ready()

The implementation of `impl_is_ready()` inserts the correct server name into the object names of the server's objects. This is not done for any object references that have already been passed out of the address space.

Normally, you set the server name by calling `impl_is_ready()`. Alternatively, you can set the server name using the method `ORB.setServerName()`.

Other interactions with OrbixWeb such as calling an operation on a remote object, or using the locator, also cause difficulties if they occur in a persistent server before `impl_is_ready()` is called.

Persistent servers, once they have called `impl_is_ready()`, behave as shared activation mode servers. In line with the CORBA specification, if a server is registered as unshared or per-method, `impl_is_ready()` fails if the server is launched manually.

ORB.connect()

The OMG standard approach to launching a persistent server is to use `org.omg.CORBA.ORB.connect()`.

Because this approach provides no way of specifying the server name, you must use *one* of the following to specify the server name:

- *Before* you connect, use `ORB.setServername()`

  or

- Add the following to the `java` or `owjava` command line:

  `-DOrbixWeb.server_name`

Unregistered Servers

In some circumstances, it may be useful not to register servers with the Implementation Repository. To support this, you can configure the OrbixWeb daemon to allow unregistered servers by using the `-u` switch. Any server process can then be started manually. When the server calls `impl_is_ready()`, it can pass any string as its server name. The daemon does not check if this is a server name known to it. Refer to "Using the Java Daemon" on page 273 for details of the `-u` switch.

A disadvantage of this approach is that an unregistered server is not known to the daemon. This means that the daemon cannot automatically invoke the Java interpreter on the server bytecode when a client binds to or invokes an operation on one of its objects. If a client invocation is to succeed, the server must be launched in advance of the invocation.

In a Java context, a more significant disadvantage of this approach is that the OrbixWeb daemon is involved in initial communications between the client and server, even though the server is not registered in the Implementation Repository. This restriction applies to all OrbixWeb servers which communicate over the standard Orbix communications protocol and limits such servers to running on hosts where an Orbix or OrbixWeb daemon process is available.

Activation Issues Specific to IIOP Servers

You do not need to register OrbixWeb servers which communicate over IIOP in the Implementation Repository. An IIOP server can publish Interoperable Object References (IORs) for the implementation objects it creates, and then await incoming client requests on those objects without contacting an OrbixWeb daemon.

Unregistered IIOP servers are important in a Java domain. This is because they can be completely independent of any supporting processes which may be platform specific. In particular, any server which relies on the `orbixd` daemon to establish initial connections depends on the availability of the daemon on specific platforms. However, you can overcome this problem by using the Java Daemon, `orbixdj`, which is platform-independent. An OrbixWeb unregistered IIOP server is completely self-contained and platform independent.

However, an IIOP server does suffer from an important disadvantage. The TCP/IP port number on which a server communicates is embedded in each IOR that a server creates. If the port is dynamically allocated to a server process on start-up, the port may differ between different processes for a single server. This may invalidate IORs created by a server if, for example, the server is killed and relaunched. OrbixWeb addresses this problem by allowing you to assign a well-known IIOP port number to the server.

These issues are discussed in detail in Chapter 9 "ORB Interoperability" on page 213.

Security Issues for OrbixWeb Servers

This section covers issues concerned with security for OrbixWeb servers. The method for addressing security issues will depend, in some cases, on which OrbixWeb daemon process you are using.

Identity of the Caller of an Operation

A server object can obtain the user name of the process that made the current operation call by using the method `get_principal()` on the `ORB` object. This method is listed in class `ORB` as follows:

```
// Java
// In package org.omg.CORBA
// in class ORB.

public org.omg.CORBA.Principal get_principal();
```

Server Security

Note: The Java Daemon (`orbixdj`) does not support access rights for user groups. An exception to this is the pseudo user group `all`.

You must *actively* grant access control rights to ensure server security. OrbixWeb maintains two access control lists for each Implementation Repository entry, as follows:

Launch	The users or groups that can launch the associated server. Users on this list, and users in groups on this list, can cause the server to be launched by invoking on one of its objects. Only these users and groups can call `impl_is_ready()` with the Implementation Repository entry's server name.
Invoke	The users and groups that can invoke operations on any object controlled by the associated server.

The entries in the access control list can be either user names or group names. There is also a pseudo group name called `all`, which can be used to implicitly add all users to an access control list. The owner of an Implementation Repository entry is always allowed to launch it and invoke operations on its objects.

The group system is determined by the underlying operating system. For example, on UNIX, a user's group membership is determined using the user's primary group along with the user's supplementary groups, as specified in the `/etc/group` file.

You can use the `chmodit` command to modify the two access control lists. However, only the owner of an Implementation Repository entry can call the `chmodit` command on it. The original owner is the user who calls the `putit` command. Subsequently, you can change the ownership using the `chownit` command.

UNIX: Effective uid/gid of Launched Servers

Note: This section does not apply to `orbixdj`.

On UNIX, the effective *uid* and *gid* of a server process launched by the OrbixWeb daemon are determined as follows:

1. If `orbixd` is not running as the `root` (super-) user, the uid and gid of every activated server process is that of `orbixd` itself.
2. If `orbixd` is run as root, it attempts to activate a server with the `uid` and `gid` of the `principal` attempting to activate the server.

   If the `principal` is unknown (not a registered user) at the local machine on which `orbixd` is running, `orbixd` attempts to run the new server with `uid` and `gid` of a standard user "`orbixusr`".
3. If there is no such standard user `orbixusr`, `orbixd` attempts to run the new server with `uid` and `gid` of a user "`nobody`".
4. If there is no such user "`nobody`", the activation fails and an exception is returned to the caller.

It is recommended that you do *not* run `orbixd` as `root`. This would allow a client running as `root` on a remote machine to launch a server with `root` privileges on a different machine. You can avoid this security risk by setting the `set-uid` bit of the `orbixd` executable and giving ownership of the executable to a user called, for example, `orbixusr` who does not have `root` privileges. Then `orbixd`, and any server launched by the daemon, do not have root privileges. Any servers that must be run with different privileges can have the `set-uid` bit set on the executable file.

Activation and Concurrency

In the per-method activation mode, or when the secondary activation modes per-client and per-client-process are used, there is no inbuilt concurrency control between the different processes created to handle operation invocations on a given object. Each resulting process must coordinate its actions as required.

Activation Information for Servers

A server can determine a number of details about how and why it was launched:

- The activation mode (shared, unshared, per-method or persistent).
- The marker name of the object that caused the server to be launched.
- The name of the method called on that object.
- The server name.

You can determine this information in a server by invoking the relevant method (defined in interface BOA) on the ORB object as follows:

Activation Mode

Use the following method to find the activation mode under which the server is registered:

```
// Server activation modes
// (defined in IE.Iona.OrbixWeb.CORBA.BOAImpl).
static final short perMethodActivationMode  = 0;
static final short unsharedActivationMode   = 1;
static final short persistentActivationMode = 2;
static final short sharedActivationMode     = 3;
static final short unknownActivationMode    = 4;

public short myActivationMode ()
   throws SystemException;
```

Marker Name

Use the following method to find the marker name of the activation object that caused this server to be launched:

```
public String myMarkerName ()
   throws SystemException;
```

The marker name for a persistent server is null.

Marker Pattern

Use the following method to find the marker pattern that caused this server to be launched:

```
public String myMarkerPattern ()
   throws SystemException;
```

Method Name

Use the following method to find the method name used to launch this server:

```
public String myMethodName ()
   throws SystemException;
```

The method name for a persistent server is null.

Server Name

Use the following method to find the server's name:

```
public String myImplementationName ()
   throws SystemException;
```

For a persistent server this is some unspecified string until `impl_is_ready()` is called.

Each of these methods raises an exception if called by a client.

IDL Interface to the Implementation Repository

The interface to the Implementation Repository, called `IT_daemon`, is defined in IDL and implemented by `orbixd`, one of the two daemon processes available in OrbixWeb. The Java Daemon, or `orbixdj`, currently implements a subset of the `IT_daemon` interface.

The IDL operations defined in `IT_daemon` are explained in the *OrbixWeb Programmer's Reference*. Differences in implementation between `orbixd` and `orbixdj` are clearly highlighted.

The UNIX utilities, such as `putit`, `catit`, and the OrbixWeb Server Manager (available on Windows 95 and Windows NT) are implemented in terms of the daemon's IDL interface.

Using the Server Manager

Note: The Server Manager is available with the OrbixWeb Professional Edition.

The Server Manager is a graphical user interface that provides much of the functionality of the OrbixWeb utilities. The Server Manager facilitates Implementation Repository management, offering functionality similar to `putit`, `rmit`, `mkdirit` and other command utilities. It also supports the activation and deactivation of servers.

Refer to the chapter "The OrbixWeb Server Manager" in the *OrbixWeb Programmer's Reference* for a description of how to use this tool.

About the Java Daemon(orbixdj)

The Java Daemon (`orbixdj`) is a Java implementation of a subset of the `IT_daemon` interface.

The functionality provided by `orbixdj` should be sufficient for the majority of applications. In cases where particular features are not supported by the Java Daemon, the `orbixd` daemon process may be used as an alternative.

Additional Java Daemon Functionality

The Java Daemon offers the great advantage of platform independence, with a significant subset of the functionality available to `orbixd`.

In addition, it offers the following:

- An *in-process* activation mode, which is more efficient in terms of resources, and quicker to start.
- A GUI console.

Limitations of the Java Daemon

The main restriction on the use of the `orbixdj` is that is supports only the shared (multiple client) activation mode.

Refer to "Scope of the Java Daemon" on page 281 for more details on the features supported by the Java Daemon.

13

The OrbixWeb Java Daemon

The OrbixWeb Java Daemon(`orbixdj`*), is a Java implementation of the* `IT_daemon` *interface. The Java Daemon administers the Implementation Repository and is responsible for the activation of servers.*

The Implementation Repository is an important component of CORBA. This holds information about servers that can be used by the ORB to activate servers on demand from clients. In previous versions of OrbixWeb, the executable `orbixd` was required to manage this repository and carry out the activation of servers. This version of OrbixWeb provides both `orbixd` and `orbixdj` executables.

A limitation of the `orbixd` executable is that it must be run on the platform for which it was built, so automatic activation of servers on other platforms is not possible. The Java Daemon fulfils the same role as the `orbixd` executable, but as it is written in Java it can be deployed on any Java platform. This extends considerably the flexibility of the server-side ORB. The executable for the Java Daemon is called `orbixdj`.

Note: The terms Java Daemon and `orbixdj` are used interchangeably throughout the OrbixWeb documentation. References to *daemon* apply to functionality supported both by `orbixd` and `orbixdj`.

Overview of the Java Daemon

The Java Daemon is responsible for transparently activating OrbixWeb servers, and re-activating servers that have exited. It is a separate process that is intended to be always active. Clients can contact the Java Daemon as follows:

- Using the `bind()` call.
- Calling an operation on an object obtained using `string_to_object` on an IOR which contains the Java Daemon's address.

The Java Daemon activates the server if it is not already active, and provides details of the activated server to the client. The client can then use these details to contact the server directly.

When a server exits and the client detects the broken connection, the client can transparently request the Java Daemon to re-activate the server. When the Java Daemon re-activates the server, the client can resume making requests of the server.

Servers can also be launched manually and register themselves with the Java Daemon. In this case, the Java Daemon only provides details of the server's location to clients, because the server does not require activation.

Features of the Java Daemon

The following are the main features of the Java Daemon (`orbixdj`):

- Cross platform operation.
- OrbixWeb server activation.
- Orbix (C++) server activation.
- In-process and out-of-process activation.
- Graphical console.
- IIOP and Orbix Protocol support.
- Compatibility with `orbixd` (both for Orbix and OrbixWeb) and IONA's GUI tools.
- Compatibility with the OrbixWeb 2 and OrbixWeb 3 Implementation Repository format.

Using the Java Daemon

The following sections discuss how to start and configure the Java Daemon, `orbixdj`.

Starting orbixdj from Windows

You can launch the Java Daemon from the OrbixWeb menu in the Windows Start menu.

Starting orbixdj from the Command Line

To launch the Java Daemon from the command line, use the following:

```
orbixdj [-inProcess] [-textConsole] [-noProcessRedirect] [-u][-V]
[-v] [-help|-?]
```

The purpose of each switch is as follows:

Switch	Effect
`-inProcess`	By default, the Java Daemon activates servers in a separate process. This is termed *out-of-process* activation. If this switch is set, the Java Daemon starts servers in a separate thread. This is termed *in-process* activation.
`-textConsole`	By default, the Java Daemon launches a GUI console. Adding this switch causes the Java Daemon to use the invoking terminal as the console.
`-noProcessRedirect`	By default the `stdout` and `stderr` streams of servers activated in a separate process are redirected to the Java Daemon console. Specifying this switch causes the output streams to be hidden.
`-u`	This allows the use of unregistered persistently launched servers.

`-V`	This prints a detailed description of the configuration the Java Daemon uses on start up. The Java Daemon then exits.
`-v`	Causes the Java Daemon to print a summary of the configuration it runs with. The Java Daemon then exits.
`-help` `-?`	Displays the switches to `orbixdj`.

Configuring the Java Daemon

Use OrbixWeb's Configuration Tool to customise the settings for the Java Daemon. The following outlines the settings in `OrbixWeb.properties` that concern the Java Daemon. It also indicates how these settings should be changed using OrbixWeb's Configuration Tool.

For more details on the Configuration Tool, refer to Chapter 4, "Getting Started with OrbixWeb Configuration" on page 53.

Settings	Effect
`IT_IMPL_IS_READY_TIMEOUT`	When an in-process server is launched, the Java Daemon waits to be informed that the server is active before allowing the causative client request to proceed. See "Guidelines for Developing in-process Servers" on page 279 for further details. The Java Daemon waits a maximum of this amount of time, specified in milliseconds. The default is `30,000` milliseconds, or `30` seconds. ***Using the Configuration Tool*** This setting is located on the **Advanced** page in the **Miscellaneous Settings** list box. Select the option in the list box and enter a value in the text box provided.

IT_IMP_REP_PATH	This is the absolute path to the Implementation Repository. ***Using the Configuration Tool*** On the **Initialization** page, in the **Impl Repository Path** field.
IT_DAEMON_SERVER_BASE	Servers that are launched in separate processes listen on their own port. This is the value of the first port, and subsequently-allocated ports increment by 1, until the IT_DAEMON_SERVER_RANGE is exceeded. At this point the port allocation wraps, starts at IT_DAEMON_SERVER_BASE, and looks for a free port. If a port cannot be allocated, a COMM_FAILURE exception is thrown. The default is 2000. ***Using the Configuration Tool*** On the **Initialization** page, in the **Start of Server port** *base* field.
IT_DAEMON_SERVER_RANGE	Refer to IT_DAEMON_SERVER_BASE. The default is 2000. ***Using the Configuration Tool*** On the **Initialization** page, in the **Start of Server port range** field.
IT_JAVA_INTERPRETER	This is the absolute path to the Java interpreter. ***Using the Configuration Tool*** On the **General** page, in the **Java Interpreter** field.

IT_DEFAULT_CLASSPATH	This is the class path the Java Daemon will use to find Java servers when launching them. You can supplement this on a per server basis using the `-addpath` parameter to `putit`. The OrbixWeb `classes` *must* be in the `CLASSPATH`. There is no default. ***Using the Configuration Tool*** On the **General** page, in the **Default Classpath** field.
IT_ORBIXD_PORT	This is the port on which the daemon listens for incoming connections. This port supports both IIOP and the Orbix Protocol. ***Using the Configuration Tool*** On the **Initialization** page, in the **OrbixWeb daemon port** field.
IT_ORBIXD_IIOP_PORT	This is a second port on which the daemon can listen for incoming connections. This port is provided to support legacy daemons which require a separate port for each protocol. ***Using the Configuration Tool*** On the **Initialization** page, in the **OrbixWeb daemon IIOP port** field.

Viewing Output Text using the Graphical Console

The Java Daemon launches a simple graphical console that displays output text streams (`stdout` and `stderr`) from the Java Daemon and launched servers. The menu items are outlined as follows:

Menu Item	Effect
File/Exit	Causes the Java Daemon to exit. If there are active servers, a prompt to exit is displayed.
Edit/Clear	Clears the content of the console window.
Tools/Threads	Outputs information about the current thread to the console window, as shown in Figure 23 on page 278.
Tools/ Garbage Collection	Causes the Java VM to run the garbage collector synchronously, and may free more up memory.
Diagnostics/Off	Sets the level of diagnostics to none. Equivalent to calling: `setDiagnostics (0)` on the ORB.
Diagnostics/Low	Sets the level of diagnostics output to the console to `LO`. Equivalent to calling: `ORB.setDiagnostics (1)`.
Diagnostics/High	Sets the level of diagnostics output to the console to `HI`. Equivalent to calling `ORB.setDiagnostics (2)`.
Diagnostics/ORB	Sets the level of diagnostics output to the console to `ORB`. Equivalent to calling: `ORB.setDiagnostics (4)`.
Diagnostics/BOA	Sets the diagnostics output to the console to `BOA`. Equivalent to calling: `ORB.setDiagnostics (8)`.
Diagnostics/Proxy	Sets the level of diagnostics output to the console to `PROXY`. Equivalent to calling: `ORB.setDiagnostics (16)`.
Diagnostics/Request	Sets the level of diagnostics output to the console to `REQUEST`. Equivalent to calling: `ORB.setDiagnostics (32)`.
Help/About	Displays the about dialog box.

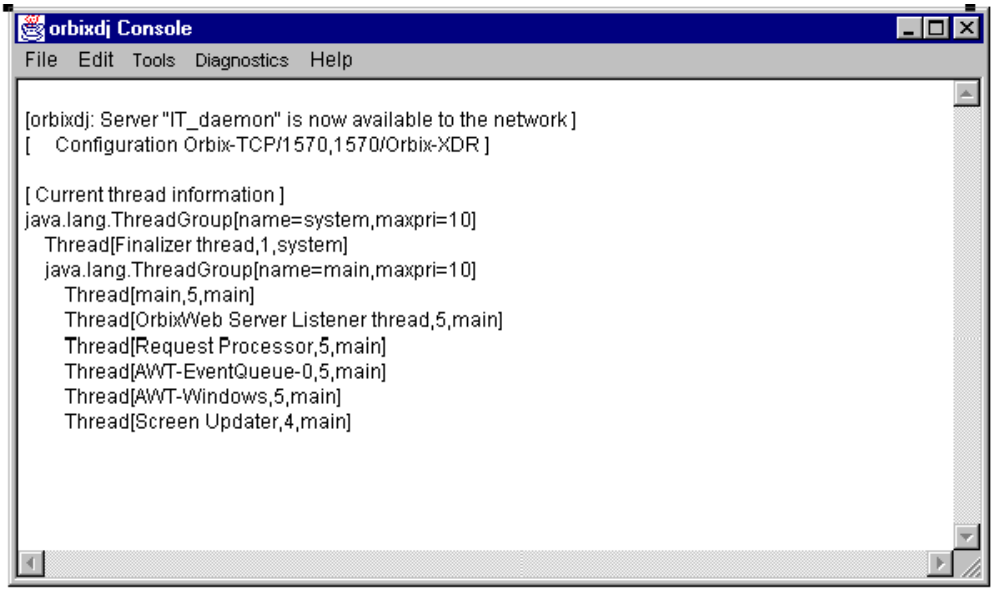

Figure 23: *Sample Output from* ***Tools/Threads*** *Menu Option*

Setting Diagnostics Levels

As with other OrbixWeb servers, you can also use the command line to specify a diagnostics level for the Java Daemon. To specify the diagnostics level on which `orbixdj` runs, use the following command:

```
-DOrbixWeb.setDiagnostics=value
```

where `value` is in the range `0-255`.

Refer to Chapter 14, "Diagnostics and Instrumentation Support" on page 285 for more details.

In-Process Activation of Servers

In-process server activation means that each launched server runs as a separate thread of execution in the daemon process. *Out-of-process* server activation means that each launched server has its own system process. The Java Daemon supports both in-process and out-of-process server activation. By default servers are activated out-of-process.

Running servers in-process rather than in a separate process brings significant benefits, particularly in scalability in terms of performance and resource consumption. These benefits include the following:

- Bind time is reduced.
- Connections are shared.
- Much less memory is required for multiple servers.

Guidelines for Developing in-process Servers

To use in-process servers, your server should initialize the ORB using:

```
IE.Iona.OrbixWeb.CORBA.ORB.init()
```

In in-process mode, this always returns the default ORB (`_CORBA.Orbix`). Currently, in-process servers do not support multiple ORBs. After the first in-process server is created, calls to `org.omg.CORBA.ORB.init()` return a `_CORBA.Orbix` object.

By their nature in-process servers are not as isolated from each other as separate processes are. Specifically, they share all global and static variables, such as the ORB itself and its object table. To prevent unintended interference between servers (including the Java Daemon itself) you need to be aware of some additional issues regarding programming of servers activated in-process. The main issues are listed as follows:

ORB Configuration

OrbixWeb configuration applies to the entire ORB. In general, you should not set configuration values in server code because this affects *all* servers in the Virtual Machine, including the Java Daemon. The capability to alter configuration values can be useful in certain situations, for example, when a different diagnostics level may be required.

Other ORB/BOA Operations

Most ORB operations apply to the entire ORB, and should be used with caution.

Exceptions to this rule for in-process activated servers are as follows:

- The operations on the OrbixWeb `OrbCurrent` object.

  You should use `OrbCurrent` to discover information about the `this` invocation.

  Refer to the description of `IE.Iona.OrbixWeb.CORBA.OrbCurrent` in the *OrbixWeb Programmer's Reference*, for more details.

- The results returned by `_OrbixWeb.ORB(ORB.init()).myServer()` and `_OrbixWeb.ORB(ORB.init()).myMarkerName()`.

The results of these operations depend on the thread they are called from (either the main server thread or the thread that has dispatched a server operation).

Other Global Objects

OrbixWeb-specific features such as filters, loaders and transformers are configured for the entire ORB. So, if you install a per-process filter in your server it is applied to all requests for all servers in the process.

The Java Daemon installs a loader and filter for its own purpose which should not be removed.

Object Table

All servers share the same object table. This object table is keyed by marker and interface type so different servers should not create objects with identical marker and interface type.

Markers should generally be assigned by the server programmer.

Server/Object Life Cycle

The Java Daemon starts up each activated server in a separate thread that calls the `main` operation of the server class. It monitors the status of this thread to determine whether the server is active or not, as indicated by `psit`.

The server becomes active when the thread calls `ORB.connect()` on instantiating a server object. It becomes inactive when the thread exits or calls `deactivate_impl()`.

Note: You *must* ensure that any clean-up operations required, such as disconnecting all server objects are performed before the thread exits. The Java Daemon does not clean up objects after the server.

The `impl_is_ready()` method is redundant for in-process servers because the Java Daemon controls event processing on behalf of the server. Refer to "Object Initialization and Connection" on page 147 to see how `impl_is_ready()` can control event-processing for out-of-process servers by changing a configuration item.

The Java Daemon security manager throws a security exception if `System.exit()` is called in a server.

Scope of the Java Daemon

The Java Daemon implements a subset of the `IT_daemon` interface. The scope of the implementation imposes some restrictions on the Java Daemon. This section discusses these restrictions and also outlines those which no longer apply.

Activation

The Java Daemon currently only supports shared server activation mode.

Java Version

The Java Daemon requires Java 1.1.

Note: The other runtime components of OrbixWeb can run on JVM 1.0.2 or JDK 1.1. This means that out-of-process servers activated by the Java Daemon can run on either JVM.

IT_daemon Interface

The Java Daemon currently implements a large subset of IONA's daemon IDL, `IT_daemon`. The following is a list of the methods which are not supported:

- `addMarker`
- `addMethod`
- `changeOwnerDir`
- `newPerMethodServer`
- `newUnSharedServer`
- `removeMarker`
- `removeMethod`
- `removeSharedMarker`
- `removeUnsharedMarker`

Utilities

The Java Daemon now supports the following utilities:

- `chmodit`
- `chownit`
- `mkdirit`
- `rmdirit`

However, because the Java Daemon only supports shared activation modes, it does not support the following switches to `putit`:

- `-per -client`
- `-per -client -pid`
- `-unshared`
- `-per -method`
- `-port`
- `-n`

- `-persistent`
- `-method`

Markers and the Implementation Repository

The only marker pattern in the Implementation Repository supported by the Java Daemon is "`*`". However, this does not prohibit the use of named markers in calls to `bind()`.

Security

The Java Daemon now supports invoke and launch access rights for users. However, access rights for user groups are not supported. An exception to this is for the pseudo group `all`.

You can use the OrbixWeb Server Manager tool and the `chmodit` command-line utility to set access rights. Refer to the *OrbixWeb Programmer's Reference* for more details.

Server Names

Because the Java Daemon now supports Implementation Repository directory utilities, it can also now support server names containing directory separator characters.

In-process Servers

In-process servers are launched using the Java Reflection API. This requires that the target class be public. If a server fails to launch when the Java Daemon is in "in-process" mode, you should ensure that the server class is public.

14

Diagnostics and Instrumentation Support

OrbixWeb provides a comprehensive diagnostics log and basic instrumentation support. Diagnostics log support is supplied by the `IE.Iona.OrbixWeb.Features.DiagnosticsLog` *API, while basic insrumentation support is provided by the* `IE.Iona.OrbixWeb.Features.InstrumentBase` *API. This chapter explains how to set diagnostics levels in OrbixWeb, and outlines the output from each diagnostics level. This chapter also discusses how to log instrumentation data in OrbixWeb.*

Setting Diagnostics

The `setDiagnostics()` method controls the level of diagnostics messages output by OrbixWeb. This method is defined in class `IE.Iona.OrbixWeb.CORBA.ORB`, as follows:

```
public int setDiagnostics(int level)
     throws org.omg.CORBA.SystemException;
```

To set diagnostics, specify the required `level` as a parameter to `setDiagnostics()`. The value of this parameter must be in the range of `0-255`.

The `setDiagnostics()` method returns the previous diagnostics level.

Diagnostics Levels

OrbixWeb provides diagnostics for specific components, each associated with a particular `level`, as follows:

level	Diagnostics Component
0	No diagnostics
1	LO
2	HI
4	ORB
8	BOA
16	PROXY
32	REQUEST
64	CONNECTION
128	DETAILED

Note: The values `LO` and `HI` correspond to the diagnostics levels `1` and `2` from earlier versions of OrbixWeb, and are included for backwards compatibility.

The `DETAILED` diagnostics component is of special significance. This controls the amount of diagnostics produced by the components. Setting the level to `DETAILED` (`128`) means that all diagnostics from the selected components are output.

Combining Diagnostics Levels

To obtain diagnostics output from particular components, add the values associated with the required components.

For example, consider obtaining detailed diagnostics associated with the BOA and REQUEST components. This involves the following steps:

1. Sum the levels associated with the BOA (8), REQUEST (32) and DETAILED components (128):

   ```
   8 + 32 + 128 =168
   ```

2. Pass the total as the `level` parameter to `setDiagnostics()`.

You can obtain full diagnostics output by setting the value to 255, the result of adding all the diagnostics components together. This produces very comprehensive output, including full buffer dumps of messages.

Overriding the Diagnostics Log

It is possible for an application to override the diagnostics log, for example, to redirect diagnostics to a file. You can override the diagnostics log by overriding the `entry()` operation implemented in `IE.Iona.OrbixWeb.Features.DiagnosticsLog`:

```
entry (ORB orb, int current_diag, int component_diag,
       Stringable component, String message,
       boolean isADetail)
```

The default `entry()` operation checks the diagnostics level, and then outputs the message to `System.out`. This message is preceded by a short string which describes the component producing the diagnostics.

To set the new diagnostics log on the ORB, use the following call:

```
myORB.setDiagnosticsLog(DiagnosticsLog l);
```

Alternative Approaches to Setting Diagnostics

You can also set the level of diagnostics output by OrbixWeb to `stdout` by:

- Using the command line.
- Using the Java Daemon graphical console.
- Using the **General** page of the OrbixWeb Configuration Tool (`owconfig`). Refer to Chapter 4, "Getting Started with OrbixWeb Configuration" on page 53.

Using the Command Line

You can use the command line to specify a diagnostic level that outputs to `stdout`; for example, by using a system parameter on start up. To specify the diagnostics level, use the following command:

```
-DOrbixWeb.setDiagnostics=value
```

where `value` is in the range `0-255`.

The diagnostics levels in this range are explained in "Setting Diagnostics" on page 286. Using the command line enables full diagnostics log support.

Using the Java Daemon Graphical Console

The Java Daemon launches a simple graphical console which displays output text streams (`stdout` and `stderr`) from the Java Daemon and launched servers. This console provides diagnostics output for each diagnostics level.

The **Diagnostics** menu item has the following options:

Menu Item	Effect
Diagnostics/Off	Sets the level of diagnostics to none. Equivalent to calling: `ORB.setDiagnostics (0)`.
Diagnostics/Low	Sets the level of diagnostics output to the console to `LO`. Equivalent to calling: `ORB.setDiagnostics (1)`.
Diagnostics/High	Sets the level of diagnostics output to the console to `HI`. Equivalent to calling: `ORB.setDiagnostics (2)`.

Diagnostics/ ORB	Sets the level of diagnostics output to the console to ORB. Equivalent to calling: `ORB.setDiagnostics(4)`.
Diagnostics/ BOA	Sets the level of diagnostics output to the console to BOA. Equivalent to calling: `ORB.setDiagnostics(8)`.
Diagnostics/ Proxy	Sets the level of diagnostics output to the console to PROXY. Equivalent to calling: `ORB.setDiagnostics(16)`.
Diagnostics/ Request	Sets the diagnostics output to the console to REQUEST. Equivalent to calling: `ORB.setDiagnostics(32)`.
Diagnostics/ Connection	Sets the diagnostics output to the console to CONNECTION. Equivalent to calling: `ORB.setDiagnostics(64)`.
Diagnostics/ Detailed	Sets the diagnostics output to the console to DETAILED. Equivalent to calling: `ORB.setDiagnostics(128)`.

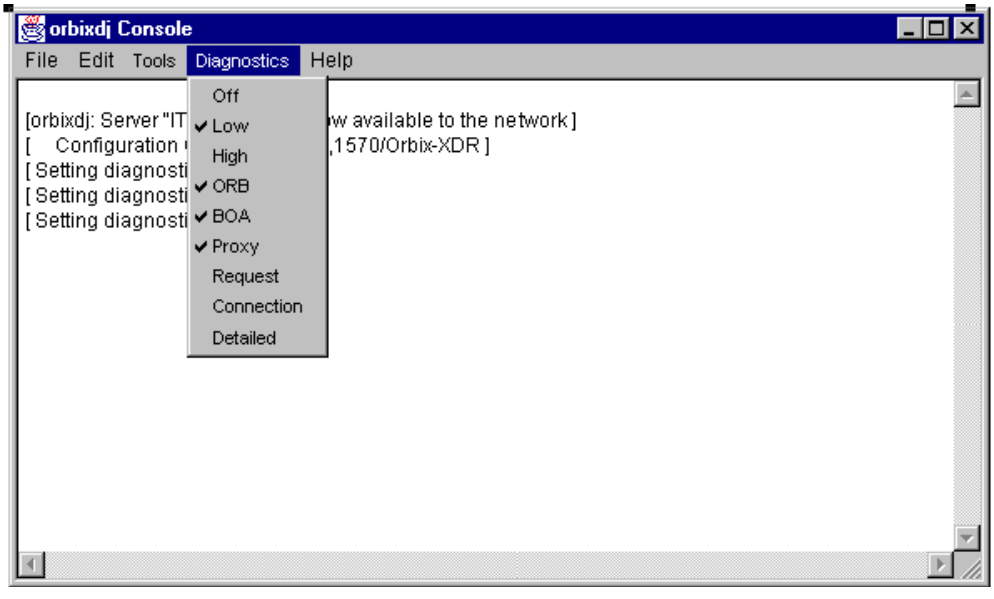

Figure 24: *Combining Diagnostics Levels Using the orbixdj Console*

Combining Diagnostics Levels

You can also use the Java Daemon graphical console to combine diagnostics levels as shown in Figure 24 on page 289.

For example, if you select the **LOW**, **ORB**, **BOA** and **Proxy** menu items, the `orbixdj` console produces a combined output for these diagnostics components.

Basic Instrumentation Support

OrbixWeb adds support for instrumentation and system management using a new class, `IE.Iona.OrbixWeb.Features.InstrumentBase`. This class provides an interface that can be implemented by application code.

InstrumentBase

The methods provided by the class `InstrumentBase` are as follows:

- `startServer (String servername, long timeout);`
- `endServer (String servername);`
- `newObj (org.omg.CORBA.Object obj);`
- `deleteObj (org.omg.CORBA.Object obj);`
- ```
  newConnection (ClientConnection conn,
                 boolean isMgmtChannel);
  ```
- `endConnection (ClientConnection conn);`
- `inRequest (org.omg.CORBA.Request obj);`
- `outReply (org.omg.CORBA.Request obj);`
- `outRequest (org.omg.CORBA.Request obj);`
- `inReply (org.omg.CORBA.Request obj);`
- `InstrGetDiagnostics (MgmtLogMessage message);`

The relevant method is called by OrbixWeb when a particular event occurs.
For example, when a server starts to process requests the `InstrumentBase.startServer()` method is called. This operation takes the server name and timeout as parameters.
  ```

You can use the `org.omg.CORBA.Object` and `org.omg.CORBA.Request` APIs to obtain additional information on the `Object` and `Request` parameters to the these methods.

Refer to the *OrbixWeb Programmer's Reference* for more details on the `org.omg.CORBA.Object` and `org.omg.CORBA.Request` APIs.

Logging Instrumentation Data

To log instrumentation data from OrbixWeb, perform the following steps:

1. Implement the required interface.
2. Register the implementation with the ORB using the following method:

```
public final void RegisterInstrumenter
   (InstrumentBase inst) {
      _instrument=inst;
}
```

Instrumentation data from OrbixWeb is now logged with the `InstrumentBase` object until the implementation is unregistered using the following method:

```
public final void UnregisterInstrumenter () {
   _instrument=null;
}
```

Data is logged by calling the interface methods with arguments of the appropriate type. For example, when a new object is connected to the object adapter the `InstrumentBase.newObj()` method is called. This operation takes the newly connected object as a parameter.

Additional Functionality

In addition, the `InstrumentBase` interface provides some extra functionality. For example, you can intercept log messages using the `InstrGetDiagnostics()` method. This method is passed a `MgmtLogMessage` object as an parameter. This contains the logged message itself as a String in the variable `content`. The `MgmtLogMessage` object also contains the diagnostics level which the message was logged at, as an integer in the variable `level`.

Refer to "Setting Diagnostics" on page 286 for details about diagnostics levels.

Note: To receive diagnostics messages, you must set the boolean variable `m_instrumentDiagnostics` to `true`.

Part IV

Topics in OrbixWeb Programming

15

Exception Handling

This chapter extends the bank example in Chapter 7 "Using and Implementing IDL Interfaces" on page 135, to raise a user-defined exception. This example shows how to throw user-defined exceptions in server code and how to handle them in a client. OrbixWeb system-defined exceptions are also described in detail.

Note: OrbixWeb does not require any special handling for exceptions. IDL exceptions are mapped to Java classes which inherit from `java.lang.Exception`. Therefore, exceptions thrown by a server can be handled by `try` and `catch` statements in the normal way.

User-Defined Exceptions

This section describes how to define exceptions in IDL and the OrbixWeb Java mapping for such user-defined exceptions.

The IDL Definitions

This section extends the interface `bank`, so that the `newAccount()` operation can raise an exception if the bank is unable to create an `account` object.

The exception `reject` is defined within the `bank` interface. It defines a string member indicating the reason that caused the `bank` to reject the request.

```
// IDL
// In file bank_demo.idl

interface account {
   readonly attribute float balance;

   void makeLodgement(in float f);
   void makeWithdrawal(in float f);
};

...

// A factory for bank accounts.
interface bank {
   exception reject { string reason; };

   account newAccount(in string name)
      raises (reject);
   ...
   // Delete an account.
   void deleteAccount(in account a);
};
```

The Generated Code

As in the example from Chapter 7, "Using and Implementing IDL Interfaces" on page 135, it is assumed that the above IDL source file is passed to the OrbixWeb IDL compiler using the following command:

```
idl -jP bank_demo bank_demo.idl
```

The IDL compiler generates Java code within the `bank_demo` package. The following Java class is generated from the IDL exception definition:

```
// Java generated by the OrbixWeb IDL compiler

package bank_demo

1   public final class reject
       extends org.omg.CORBA.UserException {
       public String reason;

       public reject() {
       }

2      public reject(String reason) {
          this.reason = reason;
       }
    }
```

1. The class `reject` (in package `bank_demo.bankPackage`) inherits from the `org.omg.CORBA.UserException`. This OrbixWeb class in turn inherits from `java.lang.Exception`. This inheritance allows `reject` to be thrown and handled as a Java exception.
2. Because the `reject` exception has one member (`reason`, of type `String`) the generated class provides a constructor that initializes this member.

The Java interface for account is generated as follows:

```
// Java generated by the OrbixWeb IDL compiler

package bank_demo;

public interface account extends org.omg.CORBA.Object {
   public float balance();

   public void makeLodgement(float f) ;
   public void makeWithdrawal(float f) ;
}
```

The generated interface for bank is as follows:

```
// Java generated by the OrbixWeb IDL compiler

package bank_demo;

public interface bank extends org.omg.CORBA.Object {
    public account newAccount(String name)
        throws bankPackage.reject;
    public void deleteAccount(account a) ;
}
```

Note: The generated method for operation newAccount() includes a throws clause for exception bank_demo.bankPackage.reject.

System Exceptions

The CORBA specification defines a set of system exceptions to which OrbixWeb adds a number of additional exceptions. These system exceptions may be raised during OrbixWeb invocations.

The standard system exceptions are implemented as a set of Java classes (in the package `org.omg.CORBA`). Each system exception is a derived class of `org.omg.CORBA.SystemException`. This in turn is a derived class of `java.lang.RuntimeException`. This means that all system exceptions can be caught in one single Java `catch` clause. The additional OrbixWeb system exceptions are implemented in the `IE.Iona.OrbixWeb.Features` package. These exceptions also inherit from the `org.omg.CORBA.SystemException` class.

A client can also handle individual system exceptions in separate `catch` clauses, as described in "Handling Specific System Exceptions" on page 301. Each system exception is implemented as a class of the following form:

```
// Java
package org.omg.CORBA;
import org.omg.CORBA.CompletionStatus;

public class <EXCEPTION TYPE>
   extends org.omg.CORBA.SystemException {
   public <EXCEPTION TYPE> (){
      ...
   }

   public <EXCEPTION TYPE> (int minor,
      CompletionStatus compl_status) {
      ...
   }

   public <EXCEPTION TYPE> (String reason) {
      ...
   }

   public <EXCEPTION TYPE> (String reason, int minor,
      CompletionStatus compl_status) {
      ...
   }
}
```

The full set of system exceptions defined by OrbixWeb are documented in an appendix to the *OrbixWeb Programmer's Reference*. This appendix provides a complete listing of the OrbixWeb system exception classes and gives a brief description of each.

The Client: Handling Exceptions

A client that calls an operation that may raise a user exception should handle that exception using an appropriate `catch` statement. Naturally, a client should also provide handlers for potential system exceptions. The following is an example client program:

```
// Java
// In file javaclient1.java

package bank_demo;

import org.omg.CORBA.SystemException;
import bank_demo.bankPackage.reject;
...

public class javaclient1 {
   public static void main(String args[]) {
   ...

      bank mybank = null;
      account currAccount = null;
      String hostname = null;
      ...

      // Search for an object offering the bank
      // server and construct a proxy.
      try {
         mybank = bankHelper.bind (":bank",hostname);

      // Create a new bank account.
         currAccount = mybank.newAccount ("chris");
      }
      catch (SystemException ex) {
         System.out.println("Unexpected system exception
1                                        :" + ex.toString());
          System.exit (1);
       }
```

```
2              catch (reject r) {
                   System.out.println("Unexpected user exception
                                                    :" + r.reason);
                   System.exit (1);
               }

               // continue here if no exception.
               ...
            }
         }
```

1. The `toString()` method defined on class `SystemException` generates a text description of the individual system exception raised.
2. The handler for the `bank_demo.bankPackage.reject` exception outputs an error message and exits the program.

Handling Specific System Exceptions

A client may also provide a handler for a specific system exception. For example, to explicitly handle a `COMM_FAILURE` exception that might be raised from a call to `bind()`, you could write the following code:

```
// Java
// In file javaclient1.java

import org.omg.CORBA.SystemException;
1  import org.omg.CORBA.COMM_FAILURE;
   ....

public class javaclient1 {
   public static void main (String args[]) {
      bank mybank = null;
      try {
         // Bind to bank with marker College_Green AIB
         // in the bank server.
         mybank = bankHelper.bind("College Green AIB:bank");
      }
      ....
2     catch (COMM_FAILURE cfe) {
         System.out.println
            ("Unexpected comm failure exception:");
```

```
            System.out.println (cfe.toString ());
            System.exit (1);
         }
         catch (SystemException se) {
            System.out.println
               ("Unexpected system exception:");
3           System.out.println ( se.toString ());
            System.exit (1);
         }
         // Continue here if no exception.
         ...
      }
   }
```

1. To handle individual system exceptions, you must import the required exceptions from the `org.omg.CORBA` package. Alternatively, you could reference the exception classes by fully scoped names.
2. The handler for a specific system exception *must* appear before the handler for `SystemException`. In Java, `catch` clauses are attempted in the order specified, and the *first* matching handler is called. A handler for `SystemException` matches all system exceptions. All system exception classes are derived classes of `SystemException` due to implicit casting.
3. If you only wish to know the type of exception that occurred, the message output by `toString()` on class `SystemException` is sufficient. A handler for an individual exception is required only when specific action is to be taken if that exception occurs.

The Server: Throwing an Exception

All OrbixWeb exceptions inherit from Java class `java.lang.Exception`. Consequently, the rules for throwing OrbixWeb exceptions follow those for throwing standard Java exceptions: you must throw an object of the exception class.

For example, you can use the following to throw an exception of IDL type `bank::reject`:

```
// Java
import bank_demo.bankPackage.reject;
...
throw new reject ("Some reason");
```

Use the automatically generated constructor of class `reject` to initialize the exception object's `reason` member with the string "`Some reason`".

The implementation of the `newAccount()` operation in class `bankImplementation` can be coded as follows:

```
// Java
// In file bankImplementation.java,

import org.omg.CORBA.SystemException;
import bank_demo.bankPackage.reject;
...

// The bank creates accounts and maintains
// a Vector of all accounts created.
class bankImplementation implements _bankOperations {
   ...
   //Internal record() operation adds new accounts to the Vector
   void record (String name, accountImplementation p)  {
      AccList.addElement (p);
      nameList.addElement (name);
   }

   // newAccount() creates a new account
   // and adds it to an account Vector.
   public account newAccount (String name)
      throws bank_demo.bankPackage.reject  {
      System.out.println ("Creating account for " + name);
      accountImplementation accImpl = null;

      // Throws a reject exception if there is an
      // existing account of the same name.
      if (nameList.contains (name))
         throw new bank_demo.bankPackage.reject
                        ("Duplicate name "+name);
      try {
         accImpl = new accountImplementation (0,name);
      }
      catch (SystemException se)  {
         System.out.println ("Exception : " + se.toString());
         System.exit (1);
      }
```

```
        account acc = new _tie_account (accImpl);
        record (name , accImpl);
        return acc;
    }
    ...
    // Account Object list.
    Vector AccList = new Vector();
    // name list
    Vector nameList = new Vector();
}
```

Operation Completion Status in System Exceptions

Class `SystemException` includes a public member variable called `status` of type `CompletionStatus`, which may be of use in some applications. This variable holds an `int` value that indicates how far the operation or attribute call progresses before the exception is raised. The return value must be one of three values defined in the OrbixWeb class `CompletionStatus` (in the package `org.omg.CORBA`).

These values are as follows:

`CompletionStatus.COMPLETED_NO`	The system exception is raised before the operation or attribute call starts to execute.
`CompletionStatus.COMPLETED_YES`	The system exception is raised after the operation or attribute call finishes its execution.
`CompletionStatus.COMPLETED_MAYBE`	It is uncertain whether or not the operation or attribute call starts execution, and, if it does, whether or not it finishes. For example, the status is `CompletionStatus.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.

16

Using Inheritance of IDL Interfaces

This chapter illustrates how to implement inheritance of IDL interfaces using OrbixWeb.

IDL allows you to define a new interface by extending the functionality provided by an existing interface, as described in Chapter 5, "Introduction to CORBA IDL" on page 69. New interfaces inherit or derive from existing interfaces. IDL also supports multiple inheritance. This allows an interface to have several immediate base interfaces.

Single Inheritance of IDL Interfaces

This section extends the bank account example, in "Defining IDL Interfaces to Application Objects" on page 136, to include current (checking) accounts:

```
// IDL
// In for example, "bank.idl".
// A bank account.
interface account {
   readonly attribute float balance;

   void makeLodgement(in float f);
   void makeWithdrawal(in float f);
};

// Derived from interface account.
interface currentAccount : account {
   readonly attribute float overdraftLimit;
   // No new operations in this example.
};

interface bank {
   exception reject { string reason; };

   account newAccount(in string name)
      raises (reject);
   void deleteAccount(in account a);
   // An extra operation:
   currentAccount newCurrentAccount
      (in string name, in float limit)
      raises (reject);
};
```

The new interface `currentAccount` derives from interface `account`. A new operation called `newCurrentAccount` is added to interface `bank` to manufacture new instances of `currentAccount`.

You could also derive from the interface `bank` to produce a new interface, for example, `commercialBank`. This would also support the `newCurrentAccount` operation.

The Client: IDL-Generated Types

As in previous chapters, it is assumed that this IDL definition has been compiled using the following command:

```
idl -jP bank_demo bank_demo.idl
```

OrbixWeb maps IDL interfaces to Java interfaces. The IDL interface inheritance hierarchy maps directly to the Java interface inheritance hierarchy, as shown in Figure 25:

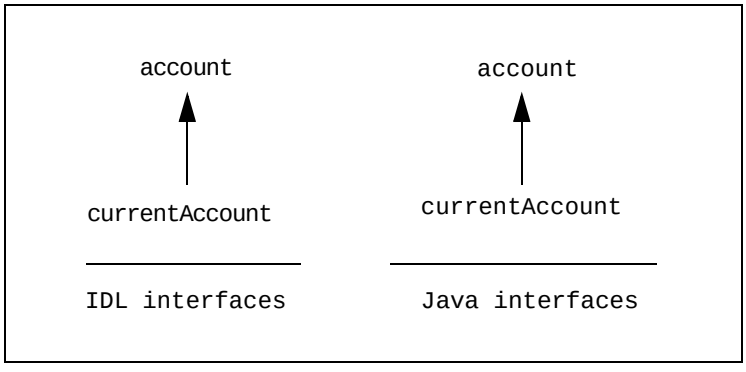

Figure 25: *IDL and Corresponding Java Hierarchies*

The interface `account` maps to the following Java interface type:

```
// Java
// Automatically generated
// in file account.java.

package bank_demo;

public interface account
   extends org.omg.CORBA.Object {

   public float balance();
   public void makeLodgement(float f);
   public void makeWithdrawal(float f);
}
```

The IDL interface `currentAccount` maps to the following interface type:

```
// Java
// Automatically generated
// In file currentAccount.java
package bank_demo;

public interface currentAccount
   extends bank_demo.account,
   org.omg.CORBA.Object {
   public float overdraftLimit();
   ...
}
```

The IDL compiler also generates Java implementation classes for the Java interfaces. These Java implementation classes provide client proxy functionality for the appropriate IDL operation. It is this proxy functionality that facilitates the distribution of objects in OrbixWeb.

In addition, the IDL compiler generates a Java *helper* class which implements the static `bind()` and `narrow()` methods.

Note: The implementation of Java interfaces in client-side generated code supplies proxy functionality to client applications. This should not be confused with the implementation of *IDL interfaces* in OrbixWeb servers.

IDL interface inheritance maps directly to the inheritance hierarchy of the generated Java interfaces, but does not map to the generated Java classes for those interfaces. Therefore, each Java class which implements an IDL generated Java interface must implement both the methods of that interface and the methods of all interfaces from which it inherits. Of course, this is an internal OrbixWeb implementation detail and does not impose any additional burden on the programmer.

This feature facilitates the mapping of IDL multiple inheritance to Java, as discussed in "Multiple Inheritance of IDL Interfaces" on page 315.

The generated Java class that implements the account interface is as follows:

```
// Java
// In file accountStub.java

package bank_demo;

import org.omg.CORBA.CompletionStatus;

public class _accountStub
   extends org.omg.CORBA.portable.ObjectImpl
   implements account {

   public float balance() {
      ...
   }

   public void makeLodgement(float f) {
      ...
   }

   public void makeWithdrawal(float f) {
      ...
   }
   ...
}
```

The generated Java class which implements the currentAccount interface is as follows:

```
// Java
// In file currentAccountStub.java

package bank_demo;
import org.omg.CORBA.CompletionStatus;

public class _currentAccountStub
   extends org.omg.CORBA.portable.ObjectImpl
   implements currentAccount {

   public float overdraftLimit() {
      ...
   }
```

```
   public float balance() {
      ...
   }

   public void makeLodgement(float f) {
      ...
   }

   public void makeWithdrawal(float f) {
      ...
   }
   ...
}
```

Using Inheritance in a Client

You can manipulate instances of `currentAccount` in a similar way to the instances of `account` in "Developing the Client Application" on page 152:

```
// Java
// In file javaclient1.java.

import org.omg.CORBA.SystemException;
import bank_demo.bankPackage.reject;
...
public class javaclient1 {
   public static void main (String args[]) {
      ...
      bank mybank = null;
      account account1 = null;
      currentAccount currAccount = null;

      try {
         // Bind to any bank object in the
         // bank server.
         mybank = bankHelper.bind (":bank");

         // Obtain a new bank account.
         account1 = mybank.newAccount ("John");

         account1.makeLodgement ((float) 56.90);
      }
```

```
            catch (reject re) {
               System.out.println
                  ("Error on newAccount():");
               System.out.println
                  ("Account creation rejected "
                     + "with reason: " + re.reason);
               System.exit(1);
            }
            catch (SystemException se) {
               System.out.println
                  ("Unexpected system exception:"
                     + se.toString ());
               System.exit(1);
            }

            try {
               // Obtain a new current account.
               currAccount = mybank.newCurrentAccount
                              ("Susie", (float) 100.00);

               currAccount.makeLodgement ((float) 87.78);
            }
            catch (reject re) {
               System.out.println (
                  "Error on newCurrentAccount():");
               System.out.println
                  ("Account creation rejected "
                     + "with reason: " + re.reason);
               System.exit(1);
            }
            catch (SystemException se) {
               System.out.println
                  ("Unexpected system exception:
                     + se.toString ());
               System.exit(1);
            }
         }
      }
```

The Server: IDL-Generated Types

This section uses the bank example to describe the two approaches to server implementation:

- The TIE Approach
- The ImplBase Approach

The TIE approach is preferred for the majority of implementations in Java. This is due to the restriction of single inheritance of classes in Java which limits the ImplBase approach. Refer to "Comparison of the ImplBase and TIE Approaches" on page 165 for a detailed discussion of both approaches.

The TIE Approach

Using the TIE approach to implementing IDL interfaces, the `currentAccount` implementation class simply implements Java interface `_currentAccountOperations`. This means that there is no implicit inheritance requirement imposed on the implementation class. This has the advantage of allowing you to inherit from any existing class that may implement a subset of the required methods.

On the server side, the IDL compiler generates the Java interface `_currentAccountOperations`. This defines the methods that a server class must implement in order to support IDL interface `currentAccount`. This Java interface inherits from type `_accountOperations`, which serves a similar purpose for IDL type `account`.

For example, if existing class `accountImplementation` implements the methods defined in interface `_accountOperations`, you could code class `currentAccountImplementation` as follows:

```
// Java
// In file currentAccountImplementation.java.
package bank_demo;
...

// Inherits account implementation, so the
// methods for type account do not need to be
// reimplemented.

class currentAccountImplementation
   extends accountImplementation,
```

```
        implements _currentAccountOperations {
        float m_limit;
        public currentAccountImpl () {
           // Details omitted.
        }

        public currentAccountImpl
           (float initialBalance, String name,float limit)
           throws SystemException {
           // Details omitted.
        }
        // Method for new IDL attribute.
        public float overdraftLimit () {
           return m_limit;
        }
     }
```

The TIE approach allows you to take advantage of the reuse characteristics of object-oriented programming.

The ImplBase Approach

The IDL compiler generates the abstract class `_currentAccountImplBase`. This supports the ImplBase approach to IDL interface implementation. To implement IDL interface `currentAccount` using the ImplBase approach, define a Java class that inherits from class `_currentAccountImplBase`, and then implement the methods defined in this class. This has important consequences for the reusability of implementation classes.

Java does not support multiple inheritance of classes. So if an existing class implements a subset of the abstract methods defined for type `currentAccount` (for example, an existing class also implements IDL type `account`), this class cannot be reused in the `currentAccount` implementation class. The `currentAccount` implementation class *must* directly implement all the operations of IDL interface `currentAccount` and all interfaces from which it inherits. This restriction severely limits the flexibility of the ImplBase approach.

Interfaces `account` and `currentAccount` can be implemented as follows:

```
// Java
// In file accountImplementation.java.
package bank_demo;

public class accountImplementation
```

```
   extends _accountImplBase {
   // Methods and variables to implement
   // interface account. As before.
   ...
}

// Java
// In file currentAccountImplementation.java.
package bank_demo;

...

// Cannot inherit account implementation,
// so the account methods must be reimplemented.
public class currentAccountImplementation
   extends _currentAccountImplBase {
   // Reimplement all methods and variables
   // defined in class accountImplementation.
   // As before.
      ...

   // Implement new methods and variables
   // for currentAccount.
   float m_limit;

   public currentAccountImpl () {
      // Details omitted.
   }

   public currentAccountImpl
      (float initialBalance, String name,
      float limit) {
      // Details omitted.
   }

   // Method for new IDL attribute
   public float overdraftLimit () {
      return m_limit;
   }
}
```

Multiple Inheritance of IDL Interfaces

IDL supports multiple inheritance of interfaces. The following serves as an example:

```
// IDL
// In for example, "bank.idl".
// A bank account.
interface account {
   readonly attribute float balance;

   void makeLodgement(in float f);
   void makeWithdrawal(in float f);
};

// Derived from interface account.
interface currentAccount : account {
   readonly attribute float overdraftLimit;
};

// Derived from interface account.
interface savingsAccount : account {
};

// Indirectly derived from interface account.
interface premiumAccount :
     currentAccount, savingsAccount {
};
```

Java also supports multiple inheritance of interfaces, but does not support multiple inheritance of classes. As in the case of single inheritance, the inheritance hierarchy of IDL interfaces maps directly to an identical inheritance hierarchy of Java interfaces which define client-side functionality. For example, the interface hierarchy in the above definition maps as shown in Figure 26 on page 316.

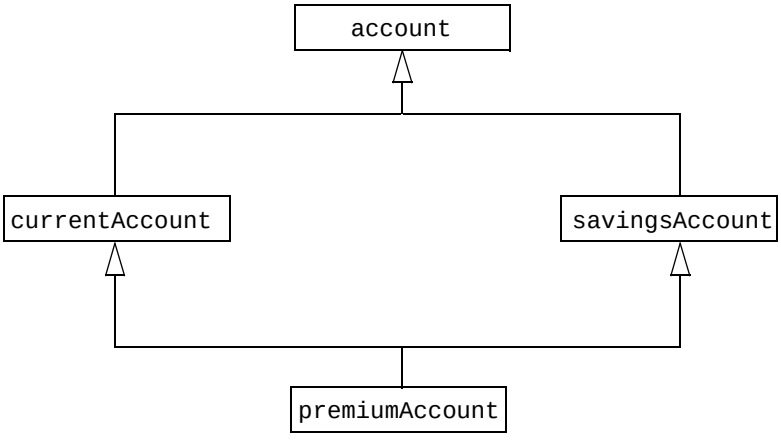

Figure 26: *Multiple Inheritance of IDL Interfaces*

The inheritance hierarchy does not map to the Java classes which implement the generated Java interfaces. Consequently, each generated Java class implements the methods of the corresponding Java interface and of all interfaces from which it inherits. In this way, a client that holds a `premiumAccount` object reference can invoke all inherited operations (from `account`, `currentAccount`, and `depositAccount`) directly on that reference.

Implementing Multiple Inheritance

On the server side, the implementation class requirements are identical to those for single inheritance.

The TIE Approach

Using the TIE approach, the implementation class must implement Java interface `_premiumAccountOperations`, but may inherit implementation methods from an existing class. However, the absence of support for multiple inheritance of classes in Java implies that a multiple inheritance hierarchy of IDL interfaces can never map directly to the implementation classes for those interfaces.

IDL avoids any ambiguity due to name clashes of operations and attributes, when two or more direct base interfaces are combined. This means that an IDL interface can not inherit from two or more interfaces with the same operation or attribute name. It is permitted, however, to inherit two or more constants, types or exceptions with the same name from more than one interface. However, you must qualify every use of these with the name of the interface, by using the full IDL scoped name.

The ImplBase Approach

Using the ImplBase approach, the implementor of type `premiumAccount` must inherit from class `_premiumAccountImplBase` and directly implement all methods for interface `premiumAccount` and all types from which it inherits.

17

Callbacks from Servers to Clients

OrbixWeb clients usually invoke operations on objects in OrbixWeb servers. However, OrbixWeb clients can implement some of the functionality associated with servers, and all servers can act as clients. This flexibility increases the range of client-server architectures you can implement with OrbixWeb. This chapter describes a common approach to implementing callbacks in an OrbixWeb application and this is illustrated by an example.

A callback is an operation invocation made from a server to an object that is implemented in a client. Callbacks allow servers to send information to clients without forcing clients to explicitly request the information.

Implementing Callbacks in OrbixWeb

This section introduces a simple model for implementing callbacks in a distributed system. The following steps are described:

- Defining the IDL interfaces for the system.
- Writing a client.
- Writing a server.

Defining the IDL Interfaces

In the example system, clients invoke operations on servers and servers invoke operations on clients. Consequently, our IDL definitions must define the interfaces through which each type of application can access the other. In the simplest case, this involves two interfaces, for example:

```
// IDL
interface ClientOps {
   ...
};

interface ServerOps {
   ...
};
```

In this model the client application supplies an implementation of type `ClientOps`, while the server implements `ServerOps`.

It is important to note that clients are *not* registered in the OrbixWeb Implementation Repository and therefore the server in this example cannot bind to the client's implementation object. Instead, our IDL definition supplies an operation that allows the client to explicitly pass an implementation object reference to the server. For example, the IDL for the example system can be defined as follows:

```
// IDL
interface ClientOps {
   void callBackToClient (in String message);
};
interface ServerOps {
   void sendObjRef (in ClientOps objRef);
};
```

"An Example Callback Application" on page 329 describes a more realistic application, and outlines the factors which you must consider when modifying this definition.

Writing a Client

The first step in writing a client is to implement the interface for the client objects, in this case type `ClientOps`. You can use the TIE or ImplBase approach, as if the client were an OrbixWeb server. In this example, it is assumed that the implementation is named `ClientOpsImplementation`.

The client `main()` method is as follows:

```
// Java

import org.omg.CORBA.ORB;
import org.omg.CORBA.SystemException;

public class Client {
   public static void main(String args[]) {
      // Initialize the ORB.
      ORB orb = ORB.init(args,null);
      // TIE approach.
      ClientOps clientImpl;
      ServerOps serverRef;

      try {
         // Instantiate implementation and proxy.
         clientImpl = new _tie_ClientOps
            (new ClientImplementation ());

         //Start a background event-processing thread
         //and connect to the runtime.
         ORB.connect(clientImpl);
         ServerRef = ServerOpsHelper.bind ();

         // Send object reference to server.
         ServerRef.sendObjRef (clientImpl);
      }
         // Process requests for 2 mins.
         try {
            Thread.sleep(1000*60*2);
         }
         catch (Exception ex){}
         orb.disconnect(clientImpl)
      catch (SystemException se) {
         System.out.println(
            "Unexpected exception:\n"
            + se.toString());
         return;
      }
}
```

The client creates an implementation object of type `ClientOpsImplementation`. It then binds to an object of type `ServerOps` in the server. At this point, the client holds an implementation object of type `ClientOps` and a proxy for an object of type `ServerOps`, as shown in Figure 27.

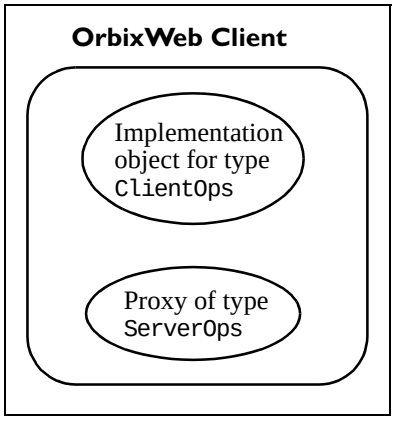

Figure 27: *Client Objects*

To allow the server to invoke operations on the `ClientOps` implementation object, the client must pass this object reference to the server. Consequently, the client now calls the operation `sendObjRef()` on the `ServerOps` proxy object, as shown in Figure 27.

The `ORB.connect()` method explicitly connects object implementations to the ORB. This method starts an event-processing thread in the background, if there is no such thread running already, the client calls `ORB.connect()` after the TIE or ImplBase object has been created. Refer to *OrbixWeb Programmer's Reference* for more details on the `connect()` method.

Finally, the client's main thread must either sleep or do other processing to avoid exiting, until it wishes to disconnect its implementation object.

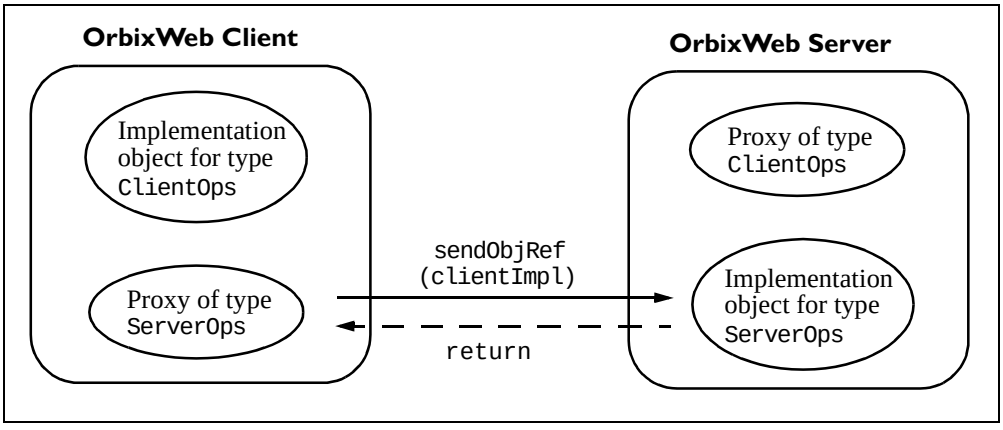

Figure 28: *Client Passes Implementation Object Reference to Server*

Writing a Server

You can code the server application as a normal OrbixWeb server. Specifically, you should define an implementation class for type `ServerOps`, and create one or more implementation objects.

The implementation of the method `sendObjRef()` for type `ServerOps` requires special attention. This method receives an object reference from the client. When this object reference enters the server address space, a proxy for the client's `ClientOps` object is created. The server will use this proxy to call back to the client. The implementation of `sendObjRef()` should store the reference to the proxy for later use.

For example, the implementation of type `ServerOps` might look as follows:

```
// Java
// (TIE approach).

public class ServerOpsImplementation
   implements _ServerOpsOperations {
   // Member variable to store proxy.
   ClientOps m_objRef;
   // Constructor.
   public ServerOpsImplementation () {
```

```
      clientObjRef = null;
   }

   // Operation implementation.
   public void sendObjRef (ClientOps objRef) {
      m_objRef = objRef;
   }
}
```

Once the server creates the proxy in its address space, it may invoke the operation `callBackToClient()`. For example, the server might initiate this call in response to an incoming event or after `impl_is_ready()` returns. The method invocation on the `ClientOps` proxy is routed to the client implementation object as shown in Figure 29.

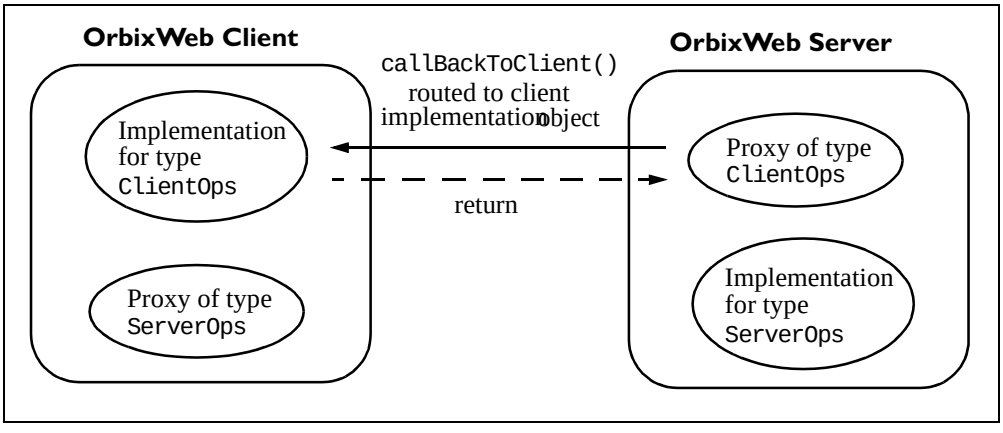

Figure 29: *Server Invokes Operation on Client's Callback Object*

The transmission of requests from server to client is possible because OrbixWeb maintains an open communications channel between client and server while both processes remain alive. The server can send the callback invocation directly to the client and does not need to route it through an OrbixWeb daemon. Therefore, the client can process the callback event without being registered in the OrbixWeb Implementation Repository and without being given a server name.

Callbacks and Bidirectional Connections

If you use the Orbix protocol, the server sends its callbacks on the same connection that the client initiated and used to make requests on the server. This means that the client does not need to accept an incoming connection.

Standard IIOP, on the other hand, requires that the client accept a connection from the server to allow the callbacks to be sent. Many firewalls do not allow an application inside the firewall to receive connections from outside. As result a client applet downloaded behind such a firewall cannot use standard IIOP to receive callbacks from a server outside the firewall.

OrbixWeb introduces an optional extension to IIOP to allow the protocol to use bidirectional connections. Bidirectional connections allow clients to receive requests from servers on the connection that the client originated to the server. This gets around the problem of downloading client applets behind a firewall. To configure your client to use bidirectional connections set the OrbixWeb configuration parameter `IT_USE_BIDIR_IIOP` to `true`. If you set this to `true`, and your server supports this feature, you can also set `IT_ACCEPT_CONNECTIONS` to `false`. This ensures that your client does not open a listening port for accepting connections. If the server does not support the feature, it attempts to open a connection back to the client according to the standard IIOP model.

Avoiding Deadlock in a Callback Model

Note: The potential for deadlock is specific to use of the OrbixWeb class `BOA` (in package `IE.Iona.OrbixWeb.CORBA`). Deadlock does not occur when the class `ORB` is used; specifically, the methods `ORB.connect()` and `ORB.disconnect()`.

When an application invokes an IDL operation on an OrbixWeb object, by default, the caller is blocked until the operation has returned. In a system where several applications have the potential to both invoke and implement operations, deadlocks may arise.

For example, in the application already described in this chapter, a simple deadlock may arise if the server attempts to call back to the client in the implementation of the method `sendObjRef()`. In this case, the client is blocked on the call to `sendObjRef()` when the server invokes `callBackToClient()`. The `callBackToClient()` call blocks the

server until the client reaches an event processing call and handles the server request. Each application is blocked, pending the return of the other, as shown in Figure 30.

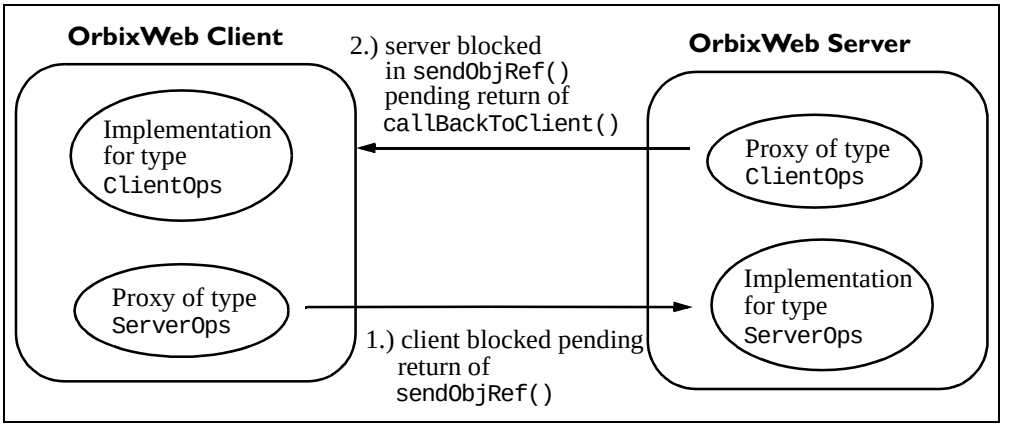

Figure 30: *Deadlock in a Simple Callback Model*

Unfortunately, it is not always possible to design a callback architecture in which simultaneous invocations between groups of processes are guaranteed never to occur. However, there are alternative methods to avoid deadlock in an OrbixWeb system. The two primary approaches are:

- Using non-blocking operation invocations.
- Using a multi-threaded event processing model.

These approaches are discussed in the two subsections which follow.

Using Non-Blocking Operation Invocations

There are two ways to invoke an IDL operation in an OrbixWeb application without blocking the caller: the first is to declare the operation as `oneway` in the IDL definition; the second is to invoke the operation using the *deferred synchronous* approach supported by the OrbixWeb Dynamic Invocation Interface (DII).

You can declare an IDL operation `oneway` only if it has no return value, `out`, or `inout` parameters. A `oneway` operation can only raise an exception if a local error occurs before

a call is transmitted. Consequently, the delivery semantics for a `oneway` request are "best-effort" only. This means that a caller can invoke a `oneway` request and continue processing immediately, but is not guaranteed that the request arrives at the server.

You can avoid deadlock, as shown in Figure 30 on page 326, by declaring either `sendObjRef()` or `callBackToClient()` as a `oneway` operation, for example:

```
// IDL
interface ClientOps {
   void callBackToClient (in String message);
};

interface ServerOps {
   oneway void sendObjRef (in ClientOps objRef);
};
```

In this case, the client's call to `sendObjRef()` returns immediately, without waiting for the server's implementation method call to return. This allows the client to enter the OrbixWeb event processing call. At this point, the callback invocation from the server is processed and routed to the client's implementation of `callBackToClient()`. When this method call returns, the server no longer blocks and both applications again wait for incoming events.

You can achieve a similar functionality by using the OrbixWeb DII deferred synchronous approach to invoking operations. As described in Chapter 19, "Dynamic Invocation Interface" on page 355, the DII allows an application to dynamically construct a method invocation at runtime, by creating a `Request` object. You can then send the invocation to the target object using one of a set of methods supported by the DII.

"Deferred Synchronous Invocations" on page 370 describes how to call the following methods on the `_CORBA.Orbix` object to invoke an operation without blocking the caller.

```
Request.send_deferred()
Request.send_oneway()
ORB.send_multiple_ requests_deferred()
ORB.send_multiple _requests_oneway()
```

If any of these methods are used, the caller can continue to process in parallel with the target implementation method. Operation results can be retrieved at a later point in the caller's processing, and avoid deadlock as if the operation call was a `oneway` invocation.

Using Multiple Threads of Execution

Note: `org.omg.CORBA.ORB.connect()` which connects an implementation to the runtime, by default also causes the ORB to launch a background event-processing thread. This means that a separate event-processing thread is not necessary. Use of the methods `processEvents()` and `processNextEvent()` outlined in this section is optional.

An OrbixWeb application may create multiple threads of execution. To avoid deadlock, it may be useful to create a separate thread dedicated to handling OrbixWeb events. For example, an OrbixWeb application could instantiate an object as follows:

```
// Java
// In file EventProcessor.java.

import IE.Iona.OrbixWeb._CORBA;
import org.omg.CORBA.SystemException;

public class EventProcessor extends Thread {
   public void run () {
      try {
            _CORBA.Orbix.processEvents
               (_CORBA.IT.INFINITE TIMEOUT)
      }
      catch (SystemException se) {
         System.out.println
            ("Unexpected exception: " + se.toString());
      }
   }
}
```

Invoking `run()` on an object of this type starts the execution of a thread that processes incoming OrbixWeb events.

If another thread in this application becomes blocked while invoking an operation on a remote object, the event processing continues in parallel. So, in the example, the remote operation can safely call back to the multi-threaded application without causing deadlock.

Event Processing Methods

OrbixWeb applications can use event processing methods that do not implicitly initialize the application server name. The client can safely call either the method `processEvents()` or the method `processNextEvent()` on the `ORB` object.

These event processing methods are defined on OrbixWeb class `BOA` (in package `IE.Iona.OrbixWeb.CORBA`). If the client is to receive callbacks, the client's `ORB` object must be initialized as type `BOA`. The client call, for example, to, `processEvents()` blocks while waiting for incoming OrbixWeb events. If the server invokes an operation on the `ClientOps` object reference forwarded by the client, this call is processed by `processEvents()` and routed to the correct method in the client's implementation object.

An Example Callback Application

The example described in this section is based on a distributed chat group application. The source code for this application is available in the `demos/WebChat` directory of your OrbixWeb installation.

Users join a chat group by downloading an OrbixWeb callback-enabled client. Using this client, the user can send text messages to a central server. The server then forwards these messages to other clients which have joined the same group.

The client provides an interface that allows each user to select a current chat group, to view messages sent to that group and to send messages to other group members. For example, if user "brian" runs the client, this user is added to the group "General" by default. At this point, the client interface appears as shown in Figure 31 on page 330.

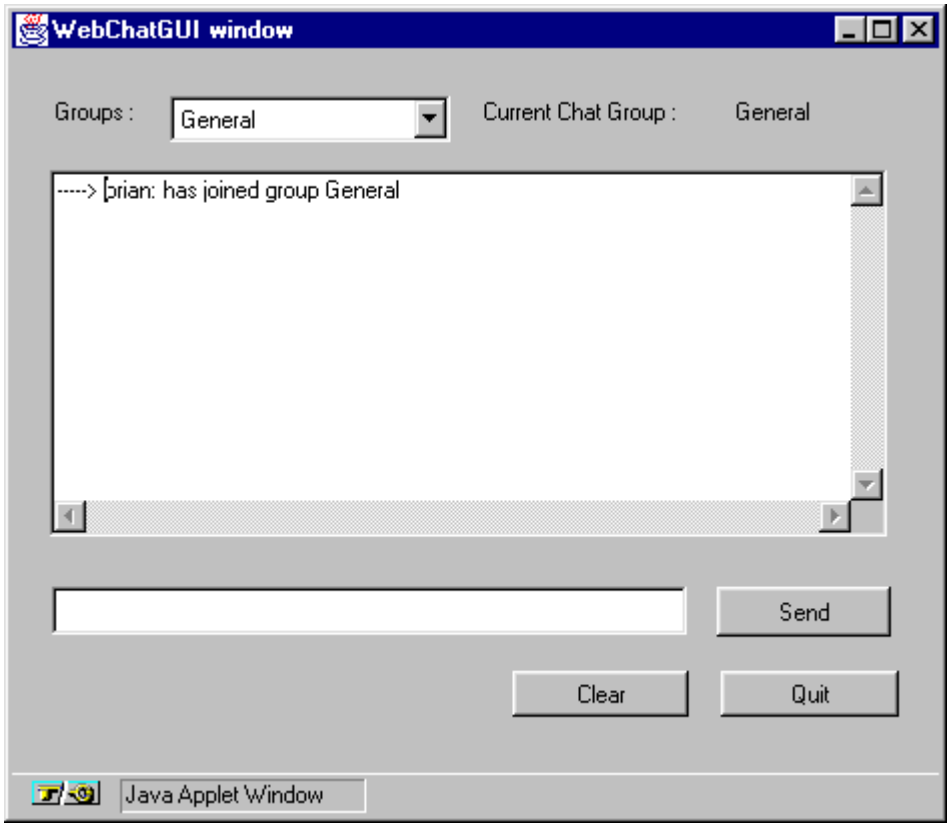

Figure 31: *WebChat Client Interface*

The **Groups** drop-down box allows the user to select a chat group. The user receives all messages sent to the current group and can only join one group at any given time.

The main text area displays all messages sent to the current group. These messages include messages from other group members and system messages indicating that other members have joined or left the group.

Finally, a text field and **Send** button allow users to send messages to the group.

The central server manages all messages sent to all chat groups. It receives the messages from client applications and forwards these messages to other clients appropriately. The server does not require any direct user interaction and can run without a user interface.

However, in this example, a server monitor interface is provided which displays statistical information about the messages in the system. This interface is shown in Figure 32.

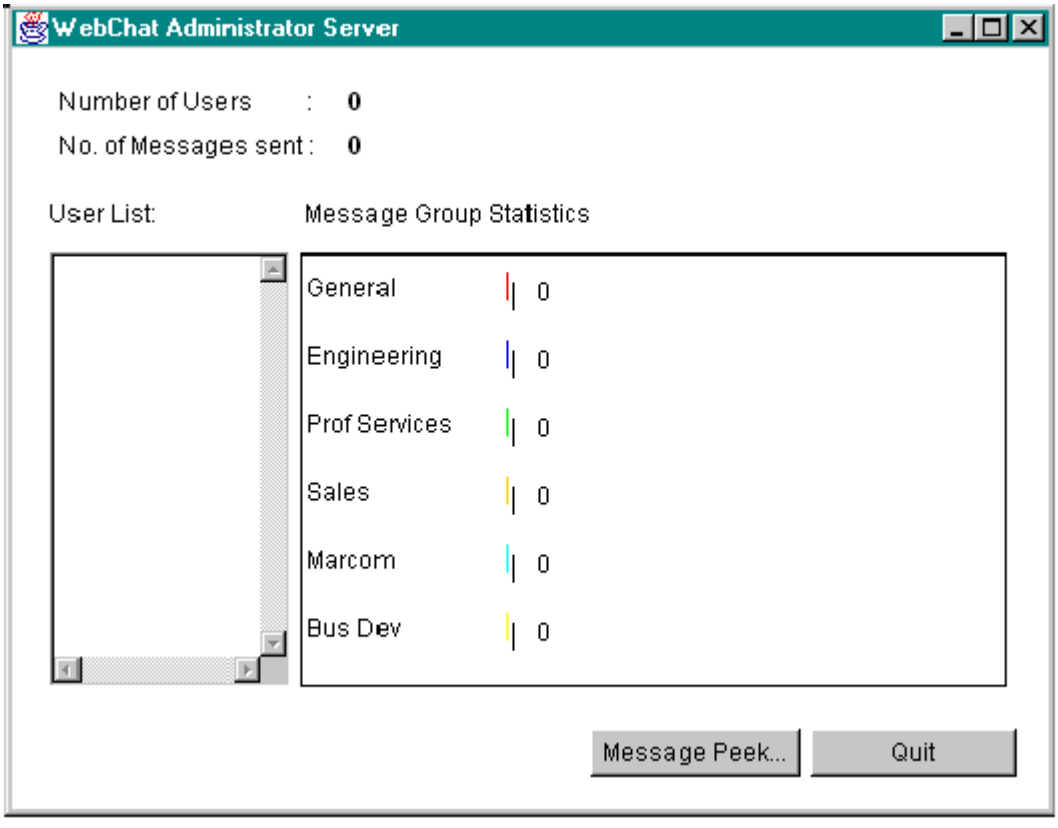

Figure 32: *WebChat Server Interface*

The interface includes information about the number of users, the members of each group, the total number of messages sent through the system and the total number of messages sent to each group. A **Message Peek** button also allows you to view each message sent through the system. This information is available because all messages are routed through this central server.

The IDL Specification

The IDL specification for this application includes two interface definitions: a `CallBack` interface implemented by clients and a `Chat` interface implemented by the server. The source code for this IDL is as follows:

```
// IDL
// In file "WebChat.idl".

// Interface definition for callbacks from
// server to client. This interface is
// implemented by clients.

interface CallBack {
   // Operation which allows the server to forward
   // a chat message to a client.
   oneway void NewMessage (in string Message);
};

// Interface which allows clients to register
// with central server. This interface is
// implemented by the server.

interface Chat {
   // Join a chat group.
   oneway void registerClient (in CallBack obj, in string Name);

   // Leave a chat group.
   oneway void RemoveClient (in CallBack obj, in string name);

   // Send a message to all group members.
   oneway void SendMessage (in string Mess);
};
```

Each client implements a single `CallBack` object. This object allows the client to receive notification from the server when new messages are sent to the client's current chat group.

The server implements a set of `Chat` objects; one object for each available chat group. A client invokes the operation `RegisterClient()` on a `Chat` object to join the chat group supported by that object. Similarly, a client application calls `RemoveClient()` to leave a chat group. A client that is registered with a chat group calls the operation `SendMessage()` to send a text message to other members of the same group.

The Client Application

You can run the `WebChatGUI` client as an applet, using the `ClientStart` applet, or as an application, using the client's `main()` method. The source code for the client application consists of the following Java classes:

- Class `local_implementation` implements the IDL interface `CallBack`.
- Class `WebChatGUI` initializes the client application and implements the client `main()` method.
- Class `Process_Events` supports the creation of a thread to handle incoming OrbixWeb events, such as callbacks from the server.

Callback Implementation

The class `local_implementation` allows a server to forward a chat message to a client. The implementation of operation `NewMessage()` displays the incoming message in the main text area of the client user interface:

```
// Java
// In file WebChatGUI.java.

package WebChat;

...

// Callback object implementation class.
class local_implementation extends _CallBackImplBase {

   WebChatGUI bkChat;

   // Callback objects hold a WebChatGUI object.
   public local_implementation(WebChatGUI bkChat) {
      super();
      this.bkChat = bkChat;
```

```
   }

   // Called by the server when a new message has been
   // sent to the current group.
   public void NewMessage(String s) {
      System.out.println
         ("Executing local_implementation::NewMessage("+s+")\n");
      try{
         bkChat.ChatEdit.appendText(s+"\n");
      }
      catch(Exception se){
         System.out.println
            ("Exception in NewMessage " + se.toString());
         System.exit(1);
      }
  }
}
```

Constructor and main() Method

The constructor of class `WebChatGUI` and the `main()` method implement the initial flow of control for the client application. The code for the `WebChatGUI` class is outlined as follows:

```
package WebChat;

import IE.Iona.OrbixWeb._CORBA;
import IE.Iona.OrbixWeb._OrbixWeb;
import IE.Iona.OrbixWeb.Features.Config;
import org.omg.CORBA.SystemException;
import org.omg.CORBA.ORB;
import java.awt.*;

// The WebChat client class.
public class WebChatGUI extends Frame {

   // WebChat constructor
   public WebChatGUI(String host, String name) {

      super("WebChatGUI window");
      // Set up WebChatGUI client window
      ...
```

```
        Host = new String(host);
        Name = new String(name);

        // Create the OrbixWeb callback object
        try {
           CallObj = new local_implementation(this);
        }
        catch (SystemException ex) {
           displayMsg ("Exception creating local implementation
                                              \n"+ ex.toString());
           System.exit(1);
        }

        // Bind to "General" group Chat object.
        try{
           TALK = ChatHelper.bind("General:WebChat",Host);
        }
        catch(SystemException se){
           displayMsg ("Exception during Bind to WebChat\n" +
                                                 se.toString());
           return;
        }

        // Register the Client with the General group server object
        try {
           TALK.RegisterClient(CallObj,Name);
           TALK.SendMessage("-----> " +Name+" : has joined group "
                                          + GroupLabel.getText());
        }
        catch (SystemException ex) {
           displayMsg("FAIL\tException during Register,
                             SendMessage \n"+ex.toString());
           System.exit(1);
        }

        // Enter the OrbixWeb event loop and wait for callbacks.
        Process_Events EventLoop = new Process_Events();
        EventLoop.start();
        show();
    }
```

```
   // WebChat client mainline used when running the client
   // as an application.
   public static void main(String args[]) {

      ORB.init(args,null);
      String hostname, username;

      // Initialize host and name from command-line
      // arguments
      ...

      // set the OrbixWeb user name
      _CORBA.Orbix.set_principal(username);

      new WebChatGUI(hostname, username);
   }
   ...
}
```

Method `RegisterClient()` invokes operation `RegisterClient()` on the server `Chat` object, passing the client's `CallBackImplementation` object reference as a parameter.

Method `Process_Events()` creates a thread in which incoming OrbixWeb events are processed, including server callback invocations. This class is defined as follows:

```
// Java
// In package WebChat,
// in class WebChatGUI.

// OrbixWeb event handler thread.
class Process_Events extends Thread {
   public Process_Events(){}

   public void run() {
   try {
   _CORBA.Orbix.processEvents
      (_CORBA.IT_INFINITE_TIMEOUT);
         // one second timeout
   }
   catch (SystemException ex) {
      ...
      return;
```

```
        }
    }
```

The definition of class `Process_Events` is as described in "Using Multiple Threads of Execution" on page 328.

The static `main()` method begins by retrieving command-line arguments and then instantiates an object of type `WebChatGUI`.

Event-Handling Methods

When the client's initialization is complete, it enters the Java event-processing loop and responds to user interface events through the method `handleEvent()` and a set of subsidiary methods. Each of the subsidiary methods handles an event for a specific user interface component. Figure 31 on page 330 shows the Web Chat client user interface.

Send Button

The **Send** button implementation sends a new message to the server object as follows:

```
// Java
// In package WebChat,
// in class WebChatGUI.

public void clickedSendButton() {
   String buff;
   buff = Name + " : " +  SendEdit.getText();
   try {
      synchronized(TALK) {
      TALK.SendMessage(buff);
   }
   catch(SystemException se){
      displayMsg
         ("Exception during SendMessage \n "+se.toString());
      System.exit(1);
   }
   SendEdit.setText("");
}
```

Clear Button

The **Clear** button implementation sets both the message and main chat group text boxes to null.

```
public void clickedClearButton() {
   SendEdit.setText("");
   ChatEdit.setText("");
}
```

Groups Drop-Down Box

The **Groups** drop-down box implementation changes groups by binding to a new server group object.

```
public void selectedGroupChoice() {
   String NewGroup = null;
   try{
      TALK.SendMessage("-----> " +Name+" : has left group "
                                    + GroupLabel.getText());
      NewGroup = new String(GroupChoice.getSelectedItem());
      GroupLabel.setText(NewGroup);

      // Remove client from current group.
      TALK.RemoveClient(CallObj,Name);

      // Bind to server object for new group.
      TALK = ChatHelper.bind(NewGroup+":WebChat",Host);

      // Register client with new group.
      TALK.RegisterClient(CallObj,Name);
      TALK.SendMessage("-----> " +Name+" : has joined group "
                                                + NewGroup);
   }
   catch(SystemException se) {
      displayMsg("Exception during SendMessage /n"
                                 + se.toString());
      System.exit(1);
   }
}
```

Quit Button

The Quit button implementation is as follows:

```
public void clickedQuitButton() {
   if (TALK!=null) {
      synchronized(TALK){
         try{
            TALK.SendMessage("-----> " +Name+" : has left
                                                WebChat");
            TALK.RemoveClient(CallObj,Name);
         }
         catch(SystemException se){
            this.hide();
            this.dispose();
            System.exit(1);
         }
      TALK=null;
   }
}
```

The Central Server Application

The server application maintains a single `Chat` implementation object for each chat group. Each `Chat` implementation object stores a list of `CallBack` proxy objects, where each proxy is associated with a single client. In this way, each server object is aware of every client which has joined that object's chat group, and can forward incoming chat messages to those group members.

The main functionality of the server is implemented in the following Java classes:

- Class `ChatImplementation` implements the IDL interface `Chat`. Each `ChatImplementation` object implements a single chat group and maintains a linked list of clients who have joined that group.
- Class `ObjectCacheEntry` implements a single entry for a linked list of client objects. Class `ChatImplementation` uses this class to store a list of `CallBack` proxy objects.
- Class `ServerGUI` initializes the server application and implements the server `main()` method.

The class ChatImplementation allows a client to register with a server object that implements a chat group. The source code for this class is as follows:

```
// Java
// In file ServerGUI.java.

package WebChat;

...

// Server-side Chat implementation class.
class ChatImplementation extends _ChatImplBase {

   // First linked list entry.
   ObjectCacheEntry firstObj;

   // Group name for current object.
   String m;

   int NoOfUsers = 0;
   static int NoMess=0;

   // Marker is implemented as group name in this example.
   ChatImplementation(String marker){
      super(marker);
      m = new String(marker);
   }

   public void SendMessage(String Mess) {
      ...
      // Update message count
      NoMess++;

      // Loop through list of registered clients.
      ObjectCacheEntry ptr = firstObj;

      while(ptr != null) {
         try{
            obj = CallBackHelper.narrow(ptr.oref);
            obj.NewMessage(Mess);
         }
         catch(SystemException se) {
         ...
```

```
      }
      ptr = ptr.next;
   }
}

public void RegisterClient(CallBack obj, String Name) {
   // Add message to server display to indicate a new
   // group mememberr
   ...
   if (firstObj == null) {
      firstObj = new ObjectCacheEntry(obj);
      return;
   }

   ObjectCacheEntry ptr = firstObj;
   while(ptr.next!=null) ptr = ptr.next;
   ptr.next = new ObjectCacheEntry(obj);
   ptr.next.prev = ptr;
}

public void RemoveClient(CallBack obj, String Name) {
   // Update main display
   ...
   // Remve callback object from list.
   if (firstObj == null) {
      ...
      return;
   }

   ObjectCacheEntry ptr = firstObj;
   CallBack tmp;

   while (ptr != null) {
      try {
         tmp =  CallBackHelper.narrow(ptr.oref);
         if ((_OrbixWeb.Object(tmp)._object_to_string()).equals
            (_OrbixWeb.Object(obj)._object_to_string())) {
            // Update linked list of objects.
            ...
            break;
         }
      }
```

```
            catch(SystemException se) {
               ...
            }
            ptr = ptr.next;
         }
      }
   }
```

An `ChatImplementation` object maintains an `ObjectCacheEntry` object as a member variable. This variable represents the head of a linked list of `CallBack` proxy objects, where each object is associated with a client that has joined the current chat group. The linked list is initially empty.

A client joins the `ChatImplementation` object's chat group by calling `RegisterClient()`. The implementation of this operation adds the client's `CallBack` object reference to the linked list. A client leaves a chat group by calling `RemoveClient()`. This removes the client's `CallBack` object reference from the linked list.

The operation `SendMessage()` allows a client to send a text message to all clients in the same chat group. The implementation of this operation accepts the message as a string parameter. It then cycles through the linked list of client object references, making a callback operation invocation on each, with the string value as a parameter. In this way, the server object redistributes text messages to all clients in a chat group.

The class `ObjectCacheEntry`, is a simple linked list node structure which stores an object reference value. The source code for this is as follows:

```
// Java
// In file ServerGUI.java.

package WebChat;

import org.omg.CORBA.*;
...

class ObjectCacheEntry {
   public ObjectCacheEntry (Object oref) {
      this.oref = oref;
   }
   // Linked list next
   public ObjectCacheEntry next;
   // Linked list previous
```

```
   public ObjectCacheEntry prev;
   public Object oref;
}
```

The class `ServerGUI` implements the flow control for the server application. The source code for this class is outlined below:

```
// Java
// In file ServerGUI.java.

package WebChat;
import IE.Iona.OrbixWeb._CORBA;
...

public class ServerGUI extends Frame {

   public static void main(String args[]) {

      ORB.init(args,null);

      mainGUI = new ServerGUI();

      // Initialize the server and enter the OrbixWeb event loop
      try {
         _CORBA.Orbix.impl_is_ready
            ("WebChat",_CORBA.IT_INFINITE_TIMEOUT);
      }
      catch(SystemException se){
         mainGUI.displayMsg
            ("Exception during impl_is_ready : " + se.toString());
         System.exit(1);
      }
      ...
   }

   // Group implementation objects.
   ChatImplementation Chat_General = null;
   ChatImplementation Chat_Engineering = null;
   ChatImplementation Chat_Marcom = null;
   ChatImplementation Chat_Sales = null;
   ChatImplementation Chat_Prof = null;
   ChatImplementation Chat_Bus = null;
```

```
   public ServerGUI() {
      super("WebChat Administrator Server");
      // Set up ServerGUI window.
      ...
      // Create the 6 server objects
      try{
         Chat_General = new ChatImplementation("General");
         Chat_Engineering = new ChatImplementation("Engineering");
         Chat_Marcom = new ChatImplementation("Marcom");
         Chat_Sales = new ChatImplementation("Sales");
         Chat_Prof = new ChatImplementation("Prof Services");
         Chat_Bus = new ChatImplementation("BusDev");
      }
      catch(SystemException se) {
         displayMsg("Exception : " + se.toString());
      }
      ...
   }
   ...
}
```

The server `main()` method first instantiates an object of type `ServerGUI`. The constructor for this object initializes the server display and creates a set of `ChatImplementation` objects. Each `ChatImplementation` object implements a single chat group, where the group name is implemented as the object marker.

When the `ServerGUI` object has been created and the server implementation objects are available, the server `main()` method invokes `impl_is_ready()` on the `_CORBA.Orbix` object and awaits incoming requests from clients.

Part V

Advanced CORBA Programming

18

Type any

This chapter gives details of the IDL type any*, and the corresponding Java class* Any *(defined in package* org.omg.CORBA*), which is used to indicate that a value of an arbitrary type can be passed as a parameter or a return value.*

Consider the following interface:

```
// IDL
interface Test {
   void op (in any a);
};
```

A client can construct an any to contain any type of value that can be specified in IDL. The client can then pass the any in a call to operation op(). An application receiving an any must determine what type of value it stores and then extract the value.

The IDL type any maps to the Java class org.omg.CORBA.Any. Refer to the *OrbixWeb Programmer's Reference* for more details. Conceptually, this class contains the following two instance variables:

- type
- value

The type is a TypeCode object that provides full type information for the value contained in the any. The Java Any class provides a type() method to return the TypeCode object. The value is the internal representation used to store Any values. The value object is accessible via the OMG standard insertion and extraction methods. These methods are described in full in this chapter.

Constructing an Any Object

You must use the ORB class (in package `org.omg.CORBA`) to construct Any objects. This is illustrated by the following example:

```
// Java

import org.omg.CORBA.*

Any a = ORB.init().create_any();
```

Inserting Values into an Any Object

The Java class Any contains a number of insertion methods that you can use to insert any of the pre-defined IDL types into an Any object. The pre-defined IDL types are as follows:

```
short
unsigned short
long
unsigned long
long long
unsigned long long
float
double
boolean
char
wchar
octet
any
Object
string
wstring
TypeCode
Principal
```

The insertion methods for these types are named `insert_short`, `insert_ushort`, `insert_long`, and so on.

A single-element insertion method simply takes the element value as a parameter. For example, the signature of `Any.insert_long()` is as follows:

```
public void insert_long(int l);
```

Helper classes for user-defined types provide `insert()` methods to support the insertion of user-defined types into an any. The signature for `insert()` can be defined as:

```
public void insert(org.omg.CORBA.Any a,
                   <user-def type> value);
```

Consider the following IDL definition:

```
// IDL

struct Foo {
   string bar;
   float number;
};

interface Flexible {
   void doit (in any a);
};
```

Assume that a client programmer wishes to pass an any containing an IDL `short` as the parameter to the `doit()` operation. The following insertion method, which is a member of class Any, may be used:

```
public void insert_short(short s);
```

The client programmer can then write the following code:

```
// Java
// Client.java

import org.omg.CORBA.*;
Flexible fRef;
Any param = ORB.init().create_any();
short toPass = 26;
try {
   fRef = FlexibleHelper.bind();

   param.insert_short (toPass);

   fRef.doit (param);
}
catch (SystemException se) {
   ...
}
```

If the client wishes to pass a more complex user-defined type, such as the struct Foo defined above, the appropriate helper class insert() methods can be used. For example, the client programmer can write the following:

```
// Java
// Client.java,

import org.omg.CORBA.*;

Flexible fRef;
Any param = ORB.init().create_any();
Foo toPass = new Foo();

toPass.bar = "Bar";
toPass.number = (float) 34.5;

try {
   fRef = FlexibleHelper.bind();

   fooHelper.insert (param, toPass);

   fref.doit (param);
}
catch (SystemException se) {
   ...
}
```

These insertion methods provide a type-safe mechanism for insertion into an any. Both the type and value of the Any are assigned at insertion. If an attempt is made to insert a value which has no corresponding IDL type, this results in a compile-time error.

Extracting Values from an Any Object

The Any Java class contains a number of methods for extracting pre-defined IDL types from an Any object. These extraction methods are named extract_long(), extract_ulong(), extract_float(), and so on. Each extraction method simply returns a value of the appropriate type.

User-defined type helper classes provide extract() methods, which support the extraction of user-defined types from an any.

The signature of this method is as follows:

```
public <user-def type>
     extract(org.omg.CORBA.Any a);
```

The following example IDL can be used to illustrate the use of extraction methods:

```
// IDL
typedef sequence<long, 10> longSeq;

interface Versatile {
   any getit();
};
```

You can extract a simple type from an any as follows:

```
// Java
// Client.java

import org.omg.CORBA.*;

Versatile vRef;
Any rv;
short toReceive;

try {
   vRef = VersatileHelper.bind();

   rv = vRef.getit();

   // extract a short value
   if ((rv.type()).kind() == TCKind.tk_short) {
      toReceive = rv.extract_short();
   }
}
catch (SystemException se) {
   ...
}
```

You can extract a sequence of type longSeq from an any as follows:

```
// Java
// Client.java

Versatile vRef;
org.omg.CORBA.Any rv;
long[] toReceive;

try {
   vRef = VersatileHelper.bind();

   rv = vRef.getit();

   // extract a sequence of longs
   if ((rv.type()).equal(longSeqHelper.type())) {
      toReceive = longSeqHelper.extract (rv);
   }
}
catch (SystemException se) {
   ...
}
```

OrbixWeb does not destroy the value of an any after extraction. You can therefore extract the value of an any more than once.

Note: The OrbixWeb-specific operations on any to extract or insert arrays are no longer supported. To insert or extract arrays, define array types in IDL and use the generated Helper class insert and extract operations.

Any as a Parameter or Return Value

The mapping for IDL any operation parameters and return values are illustrated by the following IDL operation:

```
// IDL
any op1 (in any a1, out any a2, inout any a3);
```

This IDL operation maps to the following Java method:

```
// Java
import org.omg.CORBA.Any;
import org.omg.CORBA.AnyHolder;

public Any op1 (Any a1, AnyHolder a2, AnyHolder a3);
```

Both `inout` and `out` parameters map to type `AnyHolder` as explained in "Details of Parameter Type Mappings" on page 133.

Additional Methods

In addition to the standard `Any` interface described in the `org.omg.CORBA.Any` abstract class, there are some additional methods on the actual implementation class `IE.Iona.OrbixWeb.CORBA.Any`:

- A `toString()` method.
- A `fromString()` method.
- A constructor `Any(java.lang.String)`.
- A `reset()` method
- A `copy()` method.
- A `clone()` method.
- An `equals()` method.
- A `containsType()` method.
- A `value()` accessor method.

You can use the methods `toString()` and `fromString()`, and the constructor that takes a string as an argument to maintain persistent `any` values.

To convert from a standard `org.omg.CORBA.Any` object to the actual implementation class `IE.Iona.OrbixWeb.CORBA.Any`, use the following casting operation:

```
IE.Iona.OrbixWeb._OrbixWeb.Any(org.omg.CORBA.Any a)
```

Note: The additional methods on the implementation class `IE.Iona.OrbixWeb.CORBA.Any` may not be supported in a future release of OrbixWeb.

19

Dynamic Invocation Interface

In a normal OrbixWeb client program, the IDL interfaces that the client can access are determined when the client is compiled. The Dynamic Invocation Interface (DII) allows a client to call operations on IDL interfaces that were unknown when the client was compiled.

IDL is used to describe interfaces to CORBA objects and the OrbixWeb IDL compiler generates the necessary support to allow clients to make calls to remote objects. Specifically, the IDL compiler automatically builds the appropriate code to manage proxies, to dispatch incoming requests within a server, and to manage the underlying OrbixWeb services.

Using this approach, the IDL interfaces that a client program can use are determined when the client program is compiled. Unfortunately, this is too limiting for a small but important subset of applications. These application programs and tools need to use an indeterminate range of interfaces: interfaces that perhaps were not even conceived at the time the applications were developed. Examples include browsers, gateways, management support tools and distributed debuggers.

OrbixWeb supports the CORBA *Dynamic Invocation Interface* (DII) that allows an application to issue requests for any interface, even if that interface was unknown at the time the application was compiled.

The DII allows invocations to be constructed by specifying, at runtime, the target object reference, the operation or attribute name and the parameters to be passed. A server receiving an incoming invocation request does not know whether the client that sent the request used the normal, static approach or the dynamic approach to compose the request.

Using the DII

This chapter uses the bank example to demonstrate the use of the DII. The example uses the following IDL definitions:

```
// IDL

// A bank account.
interface account {
   readonly attribute float balance;
   attribute long accountNumber;

   void makeLodgement(in float sum);
   void makeWithdrawal(in float sum,
      out float newBalance);
};

// A factory for bank accounts.
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);
};
```

You can make dynamic invocations by constructing a `Request` object and then invoking an operation on the `Request` object to make the request. Class `Request` is defined in the `org.omg.CORBA` package.

In the examples that follow, a request for the operation `newAccount()` is created, to dynamically invoke an operation whose static equivalent is:

```
// Java
bank b = bankHelper.bind();
account a;
a = b.newAccount("Chris", (float)1000.00);
```

Programming Steps for Using the DII

This chapter explains how a client can make dynamic invocations. To do so, the following steps are required:

1. Obtain an object reference.
2. Create a `Request` object using the object reference.
3. Populate the `Request` object with the parameters to the operation.
4. Invoke the request.
5. Obtain the result, if necessary.

These programming steps are described in detail later in this chapter.

Examples of Clients Using the DII

There are two common types of client program that 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 and then uses this information to construct DII requests.
- A client, such as a gateway, receives the details of a request to be made. In the case of a gateway, this may 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 there is any mismatch, the gateway receives an exception from OrbixWeb, and can report an error to the caller.

Some client programs also use the DII to call an operation with *deferred synchronous* semantics, which is not possible using normal static operation calls. Deferred synchronous calls are described in "Deferred Synchronous Invocations" on page 370.

Programming Steps: Code Example

The following code illustrates some of the programming steps using the standard `org.omg.CORBA.Request` operations:

```
// Java
// in class Client
import org.omg.CORBA.Request;
import org.omg.CORBA.Any;
...

// Initialize using either the Naming Service
// or ORB.string_to_object() details omitted
org.omg.CORBA.Object aBankObject = ....

// Create a Request
Request r = aBankObject._request("newAccount");

// Prepare the inout parameter
float ioVal = (float) 1000;

// Add the in string
r.add_in_arg().insert_string("Chris");

// Add the inout float
Any valAny =r.add_inout_arg().insert_float(ioVal);

// Add the Streamable for return value
accountHolder accountHdr = new accountHolder();
r.return_value().insert_Streamable(accountHdr);

// Invoke the Request
r.invoke ();

// Extract the inout argument
ioVal = valAny.extract_float();

// The account object ref. is now in the value member of
// the accountHdr variable.
```

To improve clarity, exception handling code is not included in this example or in most of the remaining examples in this chapter. However, developers should note that this sample code *will not compile* without the inclusion of OrbixWeb exception handling. Refer to

Chapter 15 "Exception Handling" on page 295 for details of how to handle exceptions in OrbixWeb.

This example is unrealistic since it assumes that the name of the operation (`newAccount`) is known. In practice, this information is obtained in some other way, for example from the Interface Repository.

The CORBA Approach to Using the DII

This section demonstrates how to use the DII using the OrbixWeb implementation of the classes and operations defined in the CORBA specification. A number of alternative approaches to setting up a `Request` are illustrated, all of which are CORBA-compliant.

Obtaining an Object Reference

Assume that there is already some server containing a number of objects that implement the interfaces in "Using the DII" on page 356. The first step in using the DII is to obtain an object reference of interface type `Object` (defined in package `org.omg.CORBA`) that references the target object.

If the full object reference of the target object is known in character string format, an object reference, of a type that implements `org.omg.CORBA.Object`, can be constructed to facilitate making a dynamic invocation on it. For example, you can invoke the method `string_to_object()` on the `org.omg.CORBA.ORB` object as follows:

```
// Java
import org.omg.CORBA.Object;
import org.omg.CORBA.ORB;

ORB orb = ORB.init(args, null);
Object o = orb.string_to_object (refStr);
```

In the above example, the variable `refStr` is a stringified object reference for the target object, perhaps retrieved from a file, a mail message, or an IDL operation call. Object references can also be obtained from the Naming Service. Refer to "Making Objects Available in OrbixWeb" on page 171 for further information on this topic.

Creating a Request

CORBA specifies two ways to construct a `Request` object. These are implemented in OrbixWeb using the `_request()` and `_create_request()` methods:

_request()

The method `_request()` is defined in interface `org.omg.CORBA.Object` It is declared as:

```
// Java
// in package org.omg.CORBA,
// in interface Object

import org.omg.CORBA.Request;

public Request _request(String operation);
```

This method takes a single parameter which specifies the name of the operation to be invoked on the target object.

_create_request()

There is also a _create_request() methods defined in interface Object. It is declared as:

```
// Java
// in package org.omg.CORBA,
// in interface Object

import org.omg.CORBA.Request;
import org.omg.CORBA.Context;
import org.omg.CORBA.NamedValue;
import org.omg.CORBA.NVList;

Request _create_request(Context ctx, String operation,
                        NVList arg_list,NamedValue result);
```

The use of these methods is described in the next two sections. An alternative approach to request construction is explained in "Resetting a Request Object for Reuse" on page 369.

Setting up a Request Using _request()

You can set up a request by invoking `_request()` on the target object, and specifying the name of the operation that is to be dynamically invoked. In the first attempt at constructing the request, the code is written in a verbose fashion so that the individual steps can be explained easily. A simpler, more compact, version of the same code is then shown.

The following steps are required in setting up a `Request` using the `_request()` method:

1. Obtain an object reference to the target object. The stringified object reference obtained earlier is used:

   ```
   // Java
   import org.omg.CORBA.Object;
   import org.omg.CORBA.ORB;
   import org.omg.CORBA.Request;

   ORB orb = ORB.init(args, null);
   Object o = orb.string_to_object (refStr);
   ```

2. Construct a `Request` object by calling `_request()` on the target object, as follows:

   ```
   Request request = o._request("newAccount");
   ```

3. Populate the `Request`. The most efficient and straightforward approach to populating a DII `Request` is the one used by the OrbixWeb IDL generated stubs. This approach takes advantage of the following methods in the `org.omg.CORBA.Request` class:

   ```
   import org.omg.CORBA.Any;
   import org.omg.CORBA.TypeCode;
   ...
   Any add_in_arg();
   Any add_inout_arg();
   Any add_out_arg();
   void set_return_type(TypeCode tc);
   Any return_value();
   ```

   It also uses the following insertion method in the `org.omg.CORBA.Any` class:

   ```
   import org.omg.CORBA.portable.Streamable;
   ...
   void insert_Streamable(Streamable s);
   ```

The example code using this approach appears as follows:

```
Request request = oRef._request("newAccount");

      // Insert the in parameter into the Request
      request.add_in_arg().insert_string ("Chris");

      // Insert the inout parameter:
      float ioVal = 1000.00);
      request.add_inout_arg().insert_float(ioVal);

      // Add the Streamable for return value
      accountHolder accountHdr = new accountHolder();
      request.return_value().insert_Streamable(accountHdr);

      // Invoke the Request
      request.invoke ();

      // Extract the inout argument
      ioVal = valAny.extract_float();

      // The account object ref. is now in the value member of
      // the accountHdr variable.
```

All non-primitive `inout` and `out` parameters are inserted as `Streamable` objects (those that implement `org.omg.CORBA.portable.Streamable`). All primitive `inout` and `out` parameters must be explicitly inserted and extracted using the various `Any` primitive insert and extract methods. Refer to Chapter 18, "Type any" on page 347, for more details on these methods.

Alternative approach

The following method provides an alternative approach to setting up a request.

1. First obtain an empty `NVList`, and build it to contain the parameters to the operation request.

   To create an operation list whose length is specified in the first parameter, invoke the method `create_list()` on the `org.omg.CORBA.ORB` object.

Note: If the IFR has been set up, an easier approach is to call `create_operation_list()` on `org.omg.CORBA.ORB`.
See "Using the DII with the Interface Repository" on page 367.

An `NVList` is a list of `NamedValue` elements. A `NamedValue` contains a name and a value, where the value is of type `Any` and is used in the DII to describe the arguments to a request. To obtain the `Any`, use the `value()` method defined on class `NamedValue`.

2. Using the following code as a guideline, create the `NVList` and add the `NamedValues`:

```
import org.omg.CORBA.NamedValue;
import org.omg.CORBA.NVList;
import org.omg.CORBA.Any;

import org.omg.CORBA.ARG_IN;
import org.omg.CORBA.ARG_INOUT;
...

NVList argList = ORB.init().create_list(2);
NamedValue owner = argList.add(ARG_IN.value);
owner.value().insert_string ( "Chris" );
NamedValue initBal = argList.add(ARG_INOUT.value);
initBal.value().insert_float ( 56.50 );

// Fill in name of operation and parameter values
```

The method `NVList.add()`[1] creates a `NamedValue` and adds it to the `NVList`. It returns a `NamedValue` pseudo object reference for the newly created `NamedValue`.

1. Class `NVList` also provides a method `add_value()` that takes three parameters: the name of the `NamedValue` (the formal parameter in the IDL operation); the value (of type `Any`) of the `NamedValue`; and a flag indicating the mode of the parameter. For example:

```
NamedValue owner = argList.add_value
   ("owner",ownerAny, ARG_IN.value);
NamedValue initBal = argList.add_value
   ("initialBalance", balAny, ARG_INOUT.value));
```

The parameter to `NVList.add()` can be a `Flags` object initialised with one of the following:

`ARG_IN.value`	Input parameters (IDL `in`).
`ARG_OUT.value`	Output parameters (IDL `out`).
`ARG_INOUT.value`	Input/output parameters (IDL `inout`).

You must choose the appropriate parameter that matches the corresponding formal argument.

The `NamedValues` added to the `NVList` correspond, in order, to the parameters of the operation. They must be inserted in the correct order.

3. To fully populate the request, update the `Any` contained in each `NamedValue` element of the argument list with the value that is to be passed in the operation request.

```
// Insert the parameter values into the
// NamedValues

owner.value().insert_string ("Chris");
balance.value.insert_float((float)100.00);
```

Compact Syntax

You can write the code in the last section in a more compact way by making use of the return values and the method `Request.arguments()` which returns the argument list (of type `NVList`):

```
// Java
import org.omg.CORBA.Object;
import org.omg.CORBA.ORB;
import org.omg.CORBA.Request;
import org.omg.CORBA.ARG_IN;
import org.omg.CORBA.ARG_INOUT;
...

// Obtain an object reference from
// string refStr
ORB orb = ORB.init(args, null);
Object o = orb.string_to_object (refStr);
```

```
// Create a Request object
Request request = oRef._request("newAccount");

// Insert the first parameter into the Request
(request.arguments().add (ARG_IN.value)).value())
.insert_string ("Chris");

// Insert the second parameter:
(request.arguments().add
(ARG_INOUT.value)).value())
.insert_float ((float) 1000.00);
```

Setting up a Request Using _create_request()

This section shows how to use the CORBA defined method `Request._create_request()` to create a request:

```
// Java
// in package org.omg.CORBA,
// in interface Object
public org.omg.CORBA.Request _create_request(
   org.omg.CORBA.Context ctx,
   String operation,
   org.omg.CORBA.NVList arg_list,
   org.omg.CORBA.NamedValue result);
```

The parameters of this method are as follows:

- Context object to be sent in the request.
- The name of the operation.
- The parameters to the operation (of type `NVList`).
- Location for the return value (of type `NamedValue`).
- The return value is a `Request` object which contains the new `Request` object.

The following example constructs a `Request` for operation `newAccount()`. The parameters "Chris" and `1000.00` are passed as before. The argument list is created as in "Setting up a Request Using _request()" on page 361 using `org.omg.CORBA.ORB.create_list()`.

The compact syntax is used to add the arguments to `argList` (of type `NVList`):

```
// Java
// As before allocate space for an
// NVList of length 2
import org.omg.CORBA.*;

ORB orb = ORB.init(args, null);

NVList argList = ORB.init().create_list(2);

   (argList.add(ARG_IN.value)).value())
      .insert_string ("Chris");
   // The second parameter to newAccount()

   (argList.add(ARG_INOUT.value)).value())
      .insert_float ((float) 1000.00);

// Construct a Request object with
// this information
Any a = ORB.init().create_any();
a.type(ORB.init().create_interface_tc(
   "IDL:account:1.0","account"));
NamedValue result = ORB.init().create_named_value
                                        ("", a, 0);
Context ctx = ORB.init().get_default_context();

Object o = orb.string_to_object (refStr);
Request request = o._create_request(
     ctx,
     "newAccount",
     argList,
     result)) {
   ...
}
```

Invoking a Request

Once the parameters are inserted, you can invoke the request as follows:

```
// Java
// Send Request and get the outcome
import org.omg.CORBA.SystemException;
...
try {
   request.invoke ();
   if ( request.env().exception() != null )
      throw request.env().exception();
}
catch (SystemException ex) {
   ...
}
catch ( java.lang.Exception ex ){
   ...
}
```

Note: A `Request` invocation can raise both OrbixWeb system exceptions and user-defined exceptions. To retrieve an exception raised in this manner, use `request.env().exception()`, as shown above.

Using the DII with the Interface Repository

If the programmer has obtained a description of the operation (of type `org.omg.CORBA.OperationDef`) from the Interface Repository, an alternative way to create an `NVList` is to call the operation `create_operation_list()` on the `org.omg.CORBA.ORB` object. This method fills in the elements of the `NVList`. If you use `org.omg.CORBA.ORB.create_list()` instead, you must fill the `NVList`.

The prototype of `create_operation_list()` is shown below:

```
// Java
// in package org.omg.CORBA,
// in class ORB
public NVList create_operation_list (
   org.omg.CORBA.OperationDef oper);
```

This method returns an NVList, initialised with the argument descriptions for the operation specified in operation. The returned NVList is of the correct length, with one element per argument. Each NamedValue element of the list has a valid name and valid flags which denote the argument passing mode. The value (of type Any) of the NamedValue has a valid type which denotes the type of the argument. The value of the argument is left blank. However it should be pointed out that this method performs more work than create_list() on org.omg.CORBA.ORB.

Setting up a Request to Read or Write an IDL Attribute

The DII can also be used to read and write attributes. To read the attribute balance, for example, the operation name should be set to "_get_balance". For example:

```
// Create a Request to read attribute balance
Request r = target._request ("_get_balance");
r.set_return_type(
   org.omg.CORBA.ORB.init().
   get_primitive_tc(
   org.omg.CORBA.TCKind.tk_float));
r.invoke();
float balance = r.return_value().extract_float();
```

In general, for attribute A, the operation name should be set to one of the following:

_get_A	This reads the attribute.
_set_A	This writes the attribute.

Operation Results

A request can be invoked as described in "Invoking a Request" on page 367. Once the invocation has been made, the return value and output parameters can be examined. If there are any out or inout parameters, then these parameters would be modified by the call, and no special action is required to access their values. Their values are contained in the NVList argument list which can be accessed using the method Request.arguments().

The operation's return value (if it is not void) can be accessed using the method Request.result() which returns a NamedValue.

Results can also be retrieved by using `Streamables` and the `Any.return_value()` operation. See the return value in the code in "Programming Steps: Code Example" on page 358 for details.

Interrogating a Request

The operation name and the target object's object reference of a `Request` can be determined using the methods `operation()` and `target()`, respectively.

Resetting a Request Object for Reuse

In an OrbixWeb client that uses the DII, it is often necessary to make several operation invocations. You can do this by declaring and instantiating individual `Request` objects for each invocation. However, OrbixWeb provides the method `reset()`, which allows you to reuse a `Request` variable.

The method `reset()` is called on the `Request` object and clears all of the `Request` fields, including its target object and operation name. For example, you can reuse the `Request` variable `r` in the example for an invocation of operation `makeLodgement()` as follows:

```
// Java
org.omg.CORBA.Request r =...
...
IE.Iona.OrbixWeb.CORBA.Request req
   = IE.Iona.OrbixWeb._OrbixWeb.Request(r);

req.reset ();
req.setTarget (oRef);
req.setOperation ("makeLodgement");
```

or as follows:

```
// Java
req.reset (oRef, "makeLodgement");
```

Deferred Synchronous Invocations

In addition to using the `invoke()` operation on a `Request`, OrbixWeb supports a deferred synchronous invocation mode. This allows clients to invoke on a target object and to continue processing in parallel with the invoked operation. At a later point, the client can check to see if a response is available, and if so can obtain the response. This may be useful to improve the throughput of a client, particularly in the case of long-running invocations.

Note: It is often more straightforward to start a thread that makes a normal CORBA call concurrently than to use deferred synchronous calls. They are defined by the OMG mainly for environments where threads are not available.

To use this invocation mode, call one of the following methods on the `Request`:

- `send_deferred()`
- `send_oneway`

send_deferred()

When calling method `send_deferred()` on the `Request`, the caller continues in parallel with the processing of the call by the target object. The caller can use the method `poll_response()` on the `Request` to determine whether the operation has completed and `get_response()` to determine the result. Consider the following code segment, which invokes a deferred request:

```
try {
   r.send_deferred();
}
catch(SystemException ex) {
// error handling
}
// Execute here in parallel with the call
```

The caller can perform a blocking wait for the response as follows:

```
try {
   r.get_response();
   // Extract result, etc
} catch(SystemException ex) {
```

```
   // get_response throws an exception on
   // failure/timeout
}
```

Alternatively, the caller can poll for the response as follows:

```
try {
   while(r.poll_response() == false){
   // Execute other code
   }
   // Extract result, etc
} catch(SystemException ex) { . . . . }
```

send_oneway()

You can call method `send_oneway()` can on any `Request`, however you must use this method for a `oneway` operation. The caller continues in parallel with the processing of the call by the target object.

Usage of `send_oneway()` is similar to `send_deferred()`, except that there is no response.

Multiple requests are also supported. There are two methods provided for this that can be called on an ORB. These are as follows:

- `ORB.send_multiple_requests_oneway()`
- `ORB.send_multiple_requests_deferred()`

The relevant prototypes are as follows:

```
// Java
// In class org.omg.CORBA.ORB
public void send_multiple_requests_oneway
   (Request[]  requests);
public void send_multiple_requests_deferred
   (Request[] requests);
```

The caller can perform a blocking wait for a response using the following code:

```
try {
Request r = orb.get_next_response();
   // Extract result, etc
} catch(SystemException ex) {
   ......
}
```

Alternatively the caller can call `get_response()` or `poll_response()` on an individual `Request` instance.

Using Filters with the DII

OrbixWeb allows a you to implement methods which are invoked at specified filter points in the invocation of a request, as described in "Filters" on page 415. All filter points that you implement are called during the invocation of a dynamic request.

20

Dynamic Skeleton Interface

The Dynamic Skeleton Interface (DSI) is the server-side equivalent of the DII. It allows a server to receive an operation or attribute invocation on any object, even one with an IDL interface unknown at compile time. The server does not need to be linked with the skeleton code for an interface to accept operation invocations on that interface.

Instead, a server can define a method that is informed of an incoming operation or attribute invocation. This method determines the identity of the object being invoked. The operation name and the types and values of each argument must be provided by the user. The method can then perform the task being requested by the client, and construct and return the result.

Just as the use of the DII is less common than the use of normal static invocations, the use of the DSI is less common than use of the static interface implementations. Also, clients are not aware that a server is in fact implemented using the DSI, clients simply makes IDL calls as normal.

Uses of the DSI

The DSI is explicitly designed to help you write gateways. Using the DSI, a gateway can 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. The gateway needs to know the protocol rules of non-CORBA system. However, it is the only part of the CORBA system which requires this knowledge. The rest of the CORBA system continues to make IDL calls as usual.

The IIOP protocol allows an object in one ORB to invoke on an object in another ORB. Non-CORBA systems do not have to support this protocol. One way to interface CORBA to such systems is to construct a gateway using the DSI. This gateway appears as a CORBA server containing many CORBA objects. The server uses the DSI to trap the incoming invocations and translate them into calls to the non-CORBA system. A combination of the DSI and DII allows a process to be a bi-directional gateway. The process can receive messages from the non-CORBA system and use the DII to make CORBA calls. It can use the DSI to receive requests from the CORBA system and translate these into messages in the non-CORBA system.

Another example of the use of the DSI is a server that contains a large number of non-CORBA objects that it wishes to make available to its clients. One way to achieve this is to provide an individual CORBA object to act as a front-end for each non-CORBA object. However, in some cases this multiplicity of objects may cause too much overhead.

Another way is to provide a single front-end object that can be used to invoke on any of the objects, probably by adding a parameter to each call that specifies which non-CORBA object is to be manipulated. This changes the client's view because the client would cannot invoke on each object individually, treating it as a proper CORBA object.

You can use the DSI to achieve the same space saving as that achieved when using a single front-end object. You can give clients a view that there is one CORBA object for each underlying object. The server indicates that it wishes to accept invocations on the IDL interface using the DSI, and when informed of such an invocation, it identifies the target object, the operation or attribute being called, and the parameters. It then makes the call on the underlying non-CORBA object, receives the result, and returns it to the calling client.

Using the DSI

To use the DSI you must perform the following steps in your server program:

1. Implement a class that extends the class `org.omg.CORBA.DynamicImplementation`.
2. Implement the `invoke()` and `_ids()` operations.

   The `ids()` operation is contained in the package `org.omg.CORBA.portable.ObjectImpl` which `DynamicImplementation` extends.
3. Create an object of this class and call `ORB.connect()` to connect the object to the ORB.

Creating DynamicImplementation Objects

The class `org.omg.CORBA.DynamicImplementation` is defined as follows:

```
public abstract class DynamicImplementation
   extends org.omg.CORBA.portable.ObjectImpl {
      public abstract void invoke
            (org.omg.CORBA.ServerRequest request)
   throws SystemException;
}
```

The `invoke()` method is informed of incoming operation and attribute requests. This method can use the `ServerRequest` parameter to do the following:

- Determine what operation or attribute is being invoked and on what object.
- Obtain `in` and `inout` parameters.
- Return `out` and `inout` parameters and the return value to the caller.
- Return an exception to the caller.

An implementation of the `invoke()` method is known as a *Dynamic Implementation Routine* (DIR).

The class DynamicImplementation is not visible to clients. Specifically, the interfaces used by clients do not inherit from class DynamicImplementation. If clients inherit from DynamicImplementation, the fact that the DSI is used at the server-side is not transparent to clients.

The ServerRequest Data Type

The ServerRequest object which is passed to DynamicImplementation.invoke() is created by OrbixWeb once it receives an incoming request and recognizes it as a request to be handled by the DSI.

The ServerRequest type is defined in IDL as follows:

```
// Pesudo IDL
// In module CORBA.

pseudo interface ServerRequest {
   String op_name();
   Context ctx();
   void params(NVList parms);
   any result(Any a);
   void except(Any a)
};
```

Instances of the ServerRequest interface are pseudo-objects. This means that references to these instances cannot be transmitted through IDL interfaces.

The attributes and operations of ServerRequest are described as follows:

op_name()	Gives the name of the operation being invoked.
ctx()	Returns the context associated with the call.
params()	Allows the invoke() operation to specify the types of incoming arguments.
result()	Allows the invoke() operation to return the result of an operation or attribute call to the caller.
except()	Allows the invoke() operation to return an exception to the caller.

Example of Using the DSI

To implement the Dynamic Implementation Routine (DIR), you must define a class that extends `org.omg.CORBA.DynamicImplementation`.

For example:

```
// Java
// In file javaserver1.java.
// Implementation of Dynamic Implementation Routine

package grid_dsi;

class grid_i extends org.omg.CORBA.DynamicImplementation {

   public void invoke(org.omg.CORBA.ServerRequest _req) {
      // Implementation of the invoke() method
   }
   public String[] _ids() {
      // Implementation of the _ids() method
   }
   ...
};
```

Your DSI class must contain the following methods:

- `_ids()`
- `invoke()`

_ids()

The `_ids()` method should return a list of all interfaces supported by the Dynamic Implementation Routine, as shown in the following sample code:

```
// Java
// In file javaserver1.java.
// Implementation of ids() method

public String[] _ids() {
   String[] tmp = {"IDL:grid:1.0"};
   return tmp;
}
```

invoke()

The following is an example of the DSI `invoke()` method:

```
// Java
// In file javaserver1.java.
// Implementation of invoke() method

// Simulates the operations on the grid interface using the DSI.
public void invoke(org.omg.CORBA.ServerRequest _req) {
   String _opName = _req.op_name() ;
   org.omg.CORBA.Any _ret = org.omg.CORBA.ORB.init().create_any();
   org.omg.CORBA.NVList _nvl = null;

   if(_opName.equals("set")) {
      _nvl = org.omg.CORBA.ORB.init().create_list(3);

      // Create a new any.
      org.omg.CORBA.Any n = org.omg.CORBA.ORB.init().create_any();

      // Insert the TypeCode(tk_short) into the new Any.
      n.type(org.omg.CORBA.ORB.init().get_primitive_tc
                           (org.omg.CORBA.TCKind.tk_short)) ;

      // Insert this Any into the NVList and set the flag to IN.
      _nvl.add_value(null, n, org.omg.CORBA.ARG_IN.value);

      // Create new Any, set Typecode to short, insert into NVList.
      org.omg.CORBA.Any m = org.omg.CORBA.ORB.init().create_any();
      m.type(org.omg.CORBA.ORB.init().get_primitive_tc
                           (org.omg.CORBA.TCKind.tk_short));
      _nvl.add_value(null, m, org.omg.CORBA.ARG_IN.value);

      // Create new Any, set Typecode to long, insert into NVList.
      org.omg.CORBA.Any value
                  = org.omg.CORBA.ORB.init().create_any();
      value.type(org.omg.CORBA.ORB.init().get_primitive_tc
                              (org.omg.CORBA.TCKind.tk_long));
      _nvl.add_value(null, value, org.omg.CORBA.ARG_IN.value);
```

```
      // Use params() method to marshal data into _nvl.
      _req.params(_nvl);

      // Get the value of row, col from Any row, col
      // and set this element in the array to the value.
      m_a[n.extract_short()][m.extract_short()] =
                                          value.extract_long() ;
      return;
   }

   if(_opName.equals("get")) {
      _ret = org.omg.CORBA.ORB.init().create_any();
      _nvl = org.omg.CORBA.ORB.init().create_list(2);

      org.omg.CORBA.Any n = org.omg.CORBA.ORB.init().create_any();
      ntype(org.omg.CORBA.ORB.init().get_primitive_tc
                        (org.omg.CORBA.TCKind.tk_short));
      _nvl.add_value(null, n, org.omg.CORBA.ARG_IN.value);

      org.omg.CORBA.Any m = org.omg.CORBA.ORB.init().create_any();
      m.type(org.omg.CORBA.ORB.init().get_primitive_tc
                        (org.omg.CORBA.TCKind.tk_short));
      _nvl.add_value(null, m, org.omg.CORBA.ARG_IN.value);
      _req.params(_nvl);
      int t = m_a[n.extract_short()][m.extract_short()] ;
      _ret.insert_long(t);
      _req.result(_ret);
      return;
   }

   if (_opName.equals("_get_height")) {
      _ret = org.omg.CORBA.ORB.init().create_any();
      _req.params(_nvl);
      _ret.insert_short(m_height);
      _req.result(_ret);
      return;
   }
```

```
    if (_opName.equals("_get_width")) {
        _ret = org.omg.CORBA.ORB.init().create_any();
        _req.params(_nvl);
        _ret.insert_short(m_width);
        _req.result(_ret);
        return;
    }
}
```

The complete code for this example is available in the `demos/grid_dsi` directory of your OrbixWeb installation.

21

The Interface Repository

This chapter describes the Interface Repository (IFR). This is the OrbixWeb component that provides persistent storage of IDL interfaces, modules, and other IDL types. OrbixWeb programs can query the Interface Repository at runtime to obtain information about IDL definitions.

Note: The Interface Repository is available with the Professional Edition of OrbixWeb.

The Interface Repository (IFR) enables persistent storage of IDL modules, interfaces and other IDL types. A program can browse through or list the contents of the Interface Repository. A client can also add and remove definitions from the Interface Repository using its IDL interface. Alternatively, given an object reference, an object's type and full details about that type can be determined at runtime by calling functions defined by the Interface Repository. These facilities are important for tools such as the following:

- Browsers that allow you to determine the types that 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 may need to determine the details of the object being invoked to construct the request using the DII.

- Gateways that require runtime type information about the type of objects being invoked.

OrbixWeb provides the `putidl` utility to enter definitions defined in an IDL file into the Interface Repository. This utility provides the simplest and safest way to populate the Interface Repository.

The Interface Repository also defines IDL operations to update its definitions and to enter new definitions. However, while you can write client code that populates the IFR interface database, this is complicated and requires a lot of consistency checking by the client application. It is possible to use the update operations to define interfaces and types which do not make sense. While the Interface Repository checks for such updates, it cannot prevent all incorrect updates.

Configuring the Interface Repository

The Interface Repository stores its data in the file system. You can configure the path name of its root directory using the `IT_INT_REP_PATH` entry in the OrbixWeb configuration file; or by setting the `IT_INT_REP_PATH` environment variable. The environment variable takes precedence.

An application can find the path name of its Interface Repository store by calling the following function on the `_CORBA.Orbix` object:

```
import IE.Iona.OrbixWeb._CORBA;
...
String s = _CORBA.Orbix.myIntRepPath();
```

Runtime Information about IDL Definitions

The Interface Repository maintains full details of the IDL definitions that are passed to it. A program can use the Interface Repository to browse through the set of modules and interfaces, determining the name of each module, the name of each interface and the full definition of that interface. Given a name of particular IDL definition, a program can find its full definition.

For example, given any object reference a program can use the Interface Repository to determine the following information about that interface:

- The module in which the interface was defined, if any.
- The interface name.
- The attributes of the interface, and their definitions.
- The operations of the interface, and their full definitions, including parameter, context and exception definitions.
- The base interfaces of the interface.

There is also a short example at the end of this chapter which demonstrates the use of the Interface Repository.

Using the Interface Repository

The Interface Repository is located in the `bin` directory of your OrbixWeb installation. The overall requirements for using the Interface Repository are as follows:

- You must set the `IT_INT_REP_PATH` in the OrbixWeb configuration file, or the `IT_INT_REP_PATH` environment variable; and the corresponding directory must exist.
- The Interface Repository must be installed as explained in "Installing the Interface Repository" on page 383.
- An application must import relevant Java classes.

Installing the Interface Repository

The Interface Repository is itself an OrbixWeb server. The interfaces to its objects are defined in IDL and it must be registered with the Implementation Repository. The Interface Repository can then be activated by the OrbixWeb daemon, or manually launched.

The executable file of the Interface Repository is `ifr`. This takes the following switches:

`-L`	Immediately load data from the IFR directory. The default is to load data on demand at runtime as it is required.
`-v`	Print version information about the Interface Repository.
`-h`	Print summary of switches.
`-t <time>`	Specifies the timeout in seconds for the Interface Repository server. The default is infinity.

You can explicitly run the Interface Repository executable as a background process. This has the advantage that the Interface Repository can initialize itself before any other processes need to use it, especially if you specify the `-L` switch.

The registration record in the Implementation Repository should be named "`IFR`" as follows:

```
putit IFR <absolute path name and switches>
```

To terminate the Interface Repository process, use the `killit` utility. Alternatively you can use the Windows Server Manager GUI utility or send the `SIGINT` signal (`^C`), as appropriate.

Utilities for Accessing the Interface Repository

OrbixWeb provides three utilities for accessing the IFR as follows:

- `putidl`
- `readifr`
- `rmidl`

The `putidl` utility is used to populate the IFR with definitions. `readifr` is used to check its contents, while `rmidl` removes definitions from the IFR.

putidl

The `putidl` utility is the basic means of populating the IFR. You can register the IDL definitions of, for example, `grid.idl` as follows:

```
putidl grid.idl
```

The Interface Repository stores its information in the directory specified by IT_INT_REP_PATH. The putidl utility parses the definitions in the file grid.idl and integrates the definitions into the existing definitions of the repository. If the file grid.idl uses definitions already registered in the IFR, the utility checks that the definitions are used consistently before updating the repository. If grid.idl is modified, for example by adding an extra attribute to the grid interface, it is possible to update the IFR by repeating the command putidl grid.idl. This causes old definitions to be overwritten by new definitions. putidl also checks for consistency.

While putidl takes a file as an argument, the IFR itself is not file oriented. Once the file grid.idl has been registered using putidl, the IFR has no knowledge of specifically where those definitions came from. This means that the IDL file cannot be removed from the IFR because the file, as such, does not exist within the IFR. To simplify the maintenance of definitions within the IFR it is recommended that you use modules to group definitions in a logical manner.

The syntax for the putidl command is:

```
putidl { [-?] | [-v] [-h <hostname>] [-s <filename for output>]
                                      [-I<path>] <IDL file name> }
```

These switches have the following effects:

-?	Lists the allowed switches for the putidl tool.
-v	Outputs version information for the putidl tool.
-h	Allows you to specify a host for the object.
-s	Specifies that no output from the putidl tool should be displayed.
-I	Allows searches for included IDL files in alternative directories.

rmidl

Complementary to putidl is the rmidl utility, used to depopulate the IFR. The rmidl utility removes individual definitions. It does not remove files. For example, to remove the attribute width from the grid interface use the following:

```
rmidl grid::[1]width
```

1. Note that the C++ scoping operator is used in IFR scoped names.

The use of modules is clearly advantageous when using `rmidl`. When a set of related definitions have been grouped under module `ModuleName`, you can easily remove these definitions from the IFR by executing `rmidl ModuleName`.

`rmidl` is given a scoped name identifying the definition to be deleted from the repository as its main argument. The general syntax of the `rmidl` command is:

```
rmidl {[-?] | [-h <hostname>] <scoped-name>}
```

where the scoped name has the format as described in `org.omg.CORBA.ScopedName`. For example, an attribute `balance`, defined in interface `account`, has the scoped name `account::balance`. The switches are as follows:

`-?`	Lists the allowed switches for the `rmidl` tool.
`-h`	Allows a host to be specified for the object.

Where a scoped name is given as the argument to `rmidl` you can remove the whole `grid` interface by using the command `rmidl grid`. You you should only use the `rmidl` utility to remove old or incorrect entries.

readifr

You can use the `readifr` utility to read a definition from the IFR and print it on the standard output. It also takes a scoped name as a argument and has the following general form:

```
readifr {[-?] | [-h hostname] [ <scoped-name> ]}
```

The switches are as follows:

`-?`	Lists the allowed switches for the `readifr` tool.
`-h`	Allows for a host to be specified for the object.
`-d`	Also lists derived interfaces.
`-t`	Also list `TypeCodes` (for parameters, return types and members).
`-c`	Does not prompt user for input. This is useful when `readifr` is invoked with no other argument.

You can invoke `readifr` with no arguments, in which case the default is to output the whole repository. Because the repository may be very large, you are prompted to confirm this operation.

Structure of the Interface Repository Data

The data in the Interface Repository is best viewed as a set of CORBA objects where, for each IDL type definition, one object is stored in the repository. Objects in the Interface Repository support one of the following IDL interface types, reflecting the IDL constructs they describe:

`Repository`	The type of the repository itself, in which all of its other objects are nested.
`ModuleDef`	The interface for a `ModuleDef` definition. Each module has a name and can contain definitions of any type (except `Repository`).
`InterfaceDef`	The interface for an `InterfaceDef` definition. Each interface has a name, a possible inheritance declaration, and can contain definitions of type attribute, operation, exception, typedef and constant.
`AttributeDef`	The interface for an `AttributeDef` definition. Each attribute has a name and a type, and a mode that determines whether or not it is `readonly`.
`OperationDef`	The interface for an `OperationDef` definition. Each operation has a name, a return value, a set of parameters and, optionally, `raises` and `context` clauses.
`ConstantDef`	The interface for a `ConstantDef` definition. Each constant has a name, a type and a value.
`ExceptionDef`	The interface for an `ExceptionDef` definition. Each exception has a name and a set of member definitions.
`StructDef`	The interface for a `StructDef` definition. Each struct has a name, and also holds the definition of each of its members.
`UnionDef`	The interface for a `UnionDef` definition. Each union has a name, and also holds a discriminator type and the definition of each of its members.

`EmumDef`	The interface for an `EnumDef` definition. Each `enum` has a name, and also holds its list of member identifiers.
`AliasDef`	The interface for a `typedef` statement in IDL. Each alias has a name and a type that it maps to.
`PrimitiveDef`	The interface for primitive IDL types. Objects of this type correspond to a type such as `short` and `long`, and are pre-defined within the Interface Repository.
`StringDef`	The interface for a `string` type. Each string type records its bound. Objects of this type do not have a name. If they have been defined using an IDL `typedef` statement, they have an associated `AliasDef` object. Objects of this type correspond to bounded strings.
`SequenceDef`	The interface for a `sequence` type. Each sequence type records its bound (a value of zero indicates an unbounded sequence type) and its element type. Objects of this type do not have a name. If they are defined using an IDL `typedef` statement, they have an associated `AliasDef` object.
`ArrayDef`	The interface for an `array` type. Each array type records its length and its element type. Objects of this type do not have a name. If they are defined using an IDL `typedef` statement, they have an associated `AliasDef` object. Each `ArrayDef` object represents one dimension. Multiple `ArrayDef` objects are required to represent a multi-dimensional array type.

In addition, the following abstract types (those without direct instances) are defined:

```
IRObject
IDLType
TypedefDef
Contained
Container
```

Understanding these types is the key to understanding how to use the Interface Repository. Refer to "Abstract Interfaces in the Interface Repository" on page 391 for more details.

Any object of an IDL interface type can be interrogated to determine its definitions. Interface types are organized in a logical manner according to the IDL interface. For example, each `InterfaceDef` object is said to contain objects representing the interface's constant, type, exceptions, attribute and operation definitions. The outermost object is of type `Repository`.

The containment relationships between the Interface Repository types are as follows:

A `Repository` can contain:

```
ConstantDef
TypedefDef
ExceptionDef
InterfaceDef
ModuleDef
```

A `ModuleDef` can contain:

```
ConstantDef
TypedefDef
ExceptionDef
ModuleDef
InterfaceDef
```

An `InterfaceDef` can contain:

```
ConstantDef
TypedefDef
ExceptionDef
AttributeDef
OperationDef
```

Objects of type `ModuleDef`, `InterfaceDef`, `ConstantDef`, `ExceptionDef` and `TypedefDef` can appear outside of any module, directly within a repository.

Given an object of any of the Interface Repository types, you can determine full details of that 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.

Simple Types

The Interface Repository defines the following simple IDL definitions:

```
// IDL
// In module CORBA.
typedef string Identifier;
typedef string ScopedName;
typedef string RepositoryId;
typedef string VersionSpec;

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
};
```

An `Identifier` is a simple name that identifies modules, interfaces, constants, typedefs, exceptions, attributes and operations.

A `ScopedName` gives an entity's name relative to a scope. A `ScopedName` that begins with "`::`" is an *absolute scoped name*. This is a name that uniquely identifies an entity within a repository. An example is `::finance::account::makeWithdrawal`. A `ScopedName` that does not begin with "`::`" is a *relative scoped name*. This is a name that identifies an entity relative to some other entity. An example is `makeWithdrawal` within the entity with the absolute scoped name `::finance::account`.

A `RepositoryId` is a string that uniquely identifies an object within a repository, or globally within a set of repositories if more than one is being used. An object can be a constant, exception, attribute, operation, structure, union, enumeration, alias, interface or module.

Type `VersionSpec` is used to indicate the version number of an Interface Repository object; that is, to allow the Interface Repository to distinguish two or more versions of a definition, each with the same name but with details that evolve over time. However, the Interface Repository is not required to support such versioning: it is not required to store more than one definition with any given name. The OrbixWeb Interface Repository currently does not support versioning.

Each Interface Repository object has an attribute (called `def_kind`) of type `DefinitionKind` that records the kind of the Interface Repository object. For example, the `def_kind` attribute of an `interfaceDef` object will be `dk_interface`. The enumerate constants `dk_none` and `dk_all` have special meanings when searching for objects in a repository.

Abstract Interfaces in the Interface Repository

There are five abstract interfaces defined for the Interface Repository. These are as follows:

- `IRObject`
- `IDLType`
- `TypedefDef`
- `Contained`
- `Container`

These are of key importance in understanding the basic structure of the Interface Repository and provide basic functionality for each of the concrete interface types.

Class Hierarchy and Abstract Base Interfaces

The Interface Repository defines five abstract base interfaces. These are interfaces that cannot have direct instances, and are used to define the other Interface Repository types:

`IRObject`	This is the base interface of all Interface Repository objects. Its only attribute defines the kind of an Interface Repository object.
`IDLType`	All Interface Repository interfaces that hold the definition of a type directly or indirectly inherit from this interface.
`TypedefDef`	This is the base interface for all Interface Repository types that can have names (except interfaces). These include structures, unions, enumerations and aliases (results of IDL `typedef` definitions).

`Contained`	Many Interface Repository objects can be contained within others and these all inherit from `Contained`.
`Container`	Some Interface Repository interfaces, such as `Repository`, `ModuleDef` and `InterfaceDef`, can contain other Interface Repository objects. These interfaces inherit from `Container`.

The interface hierarchy for all of the Interface Repository interfaces is shown in Figure 33.

Interface IRObject

Interface `IRObject` is defined as follows:

```
// IDL
// In module CORBA.
interface IRObject {
   // read interface
   readonly attribute DefinitionKind def_kind;

   // write interface
   void destroy ();
};
```

This is the base interface of all Interface Repository types. The attribute `def_kind` provides a simple way of determining the type of an Interface Repository object. Other than defining an attribute and operation, and acting as the base interface of other interfaces, `IRObject` plays no further role in the Interface Repository.

Modifying Objects of Type IRObject

You can delete an Interface Repository object by calling its `destroy()` operation. This also deletes any objects contained in the target object. It is an error to call `destroy()` on a `Repository` or a `PrimitiveDef` object.

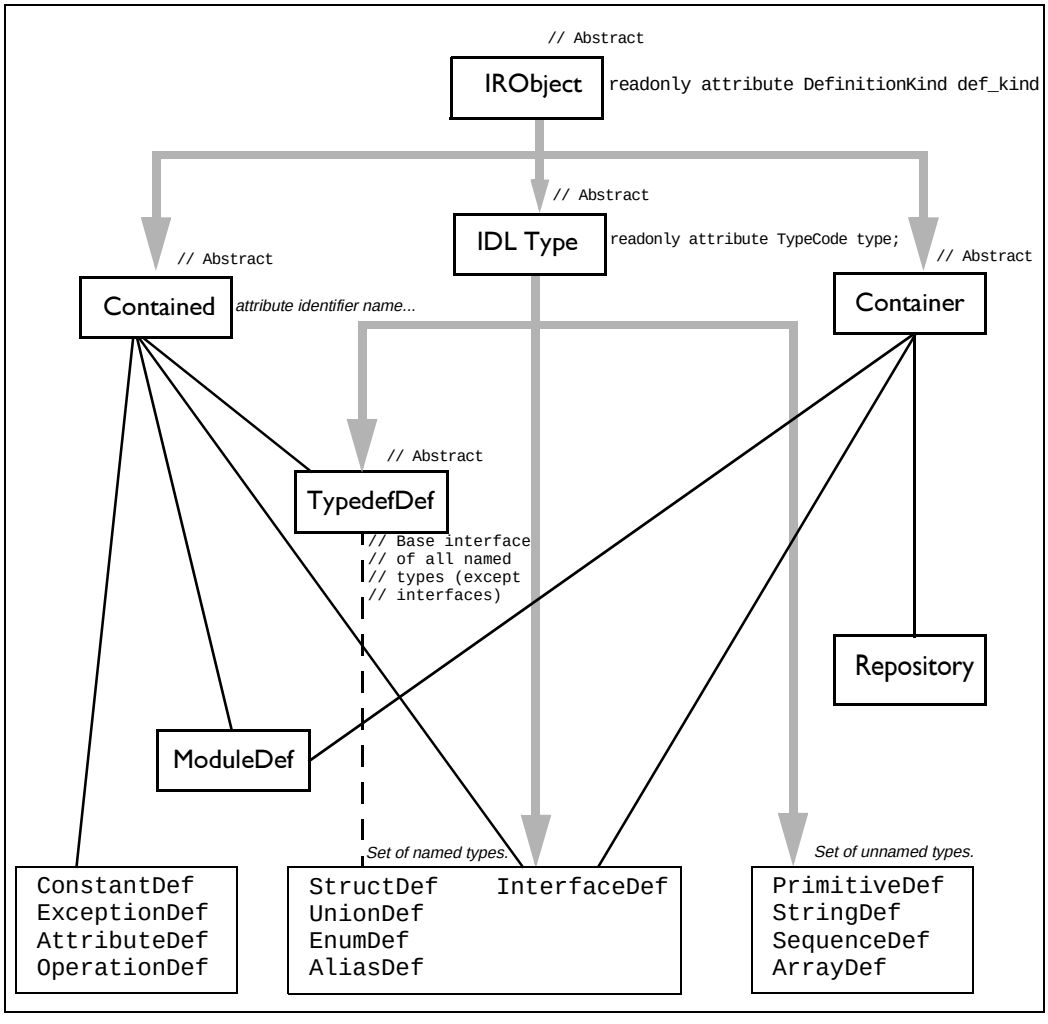

Figure 33: *Hierarchy for Interface Repository Interfaces*

Containment in the Interface Repository

Definitions in the IDL language have a nested structure. For example a module can contain definitions of interfaces and the interfaces themselves can contain definitions of attributes, operations and many others. Consider the following IDL fragment:

```
// IDL

module finance {
   interface account {
      readonly attribute float balance;
      void makeLodgement(in float amount);
      void makeWithdrawal(in float amount);
   };
   interface bank {
      account newAccount();
   };
};
```

In this example the module `finance` (represented in the Interface Repository as a `ModuleDef` object) contains two definitions: interface `bank` and interface `account` (each represented by an individual `InterfaceDef` object). These two interfaces contain further definitions. For example, the interface `account` contains a single attribute and two operations.

Since the notion of containment is basic to the structure of the IDL definitions, the Interface Repository specification abstracts the properties of containment. For example, an Interface Repository object (such as a `ModuleDef` or `InterfaceDef` object) that can contain further definitions needs a function to list its contents. Similarly, an Interface Repository object that can be contained within another Interface Repository object may want to know the identity of the object it is contained in. This leads to the definition of two abstract base interfaces, `Container` and `Contained`, which group together common operations and attributes. Most of the objects in the repository are derived from one or both of `Container` or `Contained`. The exceptions to this are instances of `PrimitiveDef`, `StringDef`, `SequenceDef` and `ArrayDef`.

You can access much of the structure of the Interface Repository by using the operations and attributes of `Container` and `Contained`. Understanding containment is the key to understanding most Interface Repository functionality.

There are three different kinds of interface which use containment. There are interfaces that inherit only from `Container`, interfaces that inherit from both `Container` and `Contained`, and interfaces that inherit only from `Contained`. These are as follows:

base `Container`	`Repository`
base `Container` and `Contained`	`ModuleDef`, `InterfaceDef`
base `Contained`	`ConstantDef`, `ExceptionDef`, `AttributeDef`, `OperationDef`, `StructDef`, `UnionDef`, `EnumDef`, `AliasDef`, `TypedefDef`

The last interface `TypedefDef` is exceptional because it is an abstract interface.

The `Repository` itself is the only interface that can be a pure `Container`. There is only one `Repository` object per Interface Repository server. This has all the other definitions nested within it.

Objects of type `ModuleDef` and `InterfaceDef` can create additional layers of nesting, and therefore these derive from both `Container` and `Contained`.

The remaining types of object have a simpler structure and derive from `Contained` only.

The Contained Interface

This section is limited to a discussion of the basic attributes and operations of interface `Contained`. Refer to the *OrbixWeb Programmer's Reference* for a full description of this interface. An outline of the `Contained` interface is as follows:

```
//IDL

typedef Identifier string;

interface Contained : IRObject {
   // Incomplete list of operations and attributes...
   ...
   attribute Identifier name;
   ...
   readonly attribute Container defined_in;
   ...
   struct Description {
      DefinitionKind kind;
```

```
      any value;
   };
   Description describe();
};
```

A basic attribute of any `Contained` object is its `name`. The attribute `name` has the type `Identifier` which is a `typedef` for a string. For example, the module `finance` is represented in the repository by a `ModuleDef` object. The inherited `ModuleDef::name` attribute resolves to the string "`finance`". Similarly, an `OperationDef` object representing `makeWithdrawal` has an `OperationDef::name` which resolves to "`makeWithdrawal`". The `Repository` object itself has no `name` because it does not inherit from `Contained`.

Another basic attribute is `Contained::defined_in`, which returns an object reference to the `Container` in which the object is defined. This attribute is all that is needed to express the idea of containment for a `Contained` object. Since a given definition appears only once in IDL, the attribute `defined_in` returns a uniquely-defined `Container` reference. However, because of the possibility of inheritance between interfaces, a given object can be contained in more than one interface. For example, interface `currentAccount` is derived from interface `account` as follows:

```
//IDL
// in module finance
interface currentAccount : account {
   readonly attribute overDraftLimit;
};
```

Here the attribute `balance` is contained in interface `account` and also contained in interface `currentAccount`. However, querying `AttributeDef::defined_in` for the `balance` attribute always returns an object for `account`. This is because the definition of attribute `balance` appears in the base interface `account`.

The operation `Contained::describe()` returns a generic `Description` structure. This provides access to details such as the parameters and return types associated with a specified object.

The Container Interface

Some of the basic definitions for interface `Container` are as follows:

```
//IDL
typedef sequence<Contained> ContainedSeq;
enum DefinitionKind {dk_name, 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};

interface Container : IRObject {
   // Incomplete list of operations and attributes
   ...
   ContainedSeq contents(
      in DefinitionKind limit_type,
      in boolean exclude_inherited);
   ...
};
```

contents()

The `contents()` operation is the most basic operation associated with a `Container`. This returns a sequence of `Contained` objects belonging to the `Container`. Using `contents` you can browse a `Container` and descend nested layers of containment. Once the appropriate `Contained` object is found, you can find the details of its definition by invoking `Contained::describe()` to obtain a detailed `Description` of the object. Using `Container::contents()` coupled with `Contained::describe()` provides a basic way of browsing the Interface Repository.

However, there are other approaches to browsing the Interface Repository which may be more efficient. These more sophisticated search operations are discussed in "Retrieving Information from the Interface Repository" on page 405.

The arguments to the `contents()` operation make use of `DefinitionKind`. This is an `enum` type which is used to tag the different kinds of repository objects. In addition to the interfaces for concrete repository objects there are three additional tags:

`dk_none`	This tag matches no repository object.
`dk_all`	This tag matches any repository object.

dk_Typedef	This tag matches any one of dk_Alias, dk_Struct, dk_Union, dk_Enum.

The parameters to contents are as follows:

limit_type	A tag of type DefinitionKind which can be used to limit the list of contents to certain kinds of repository objects. A value of dk_all lists all objects.
exclude_inherited	This argument is only relevant if the Container happens to be an InterfaceDef object. In this case, it determines whether or not inherited definitions should be included in the contents listing. true indicates they should be excluded, while false indicates they should be included.

The value returned from the contents() operation is a sequence of Contained objects which match the given criteria.

Containment Descriptions

The containment framework reveals which definitions are made within a specific interface or module. However, each interface repository object, besides being a Contained or Container, also contains the details of an IDL definition. Calling describe() on a Contained object returns a Description struct holding these details.

Both interfaces Contained and Container define their own version of a Description struct. These are, respectively, Contained::Description and Container::Description. The structure of Container::Description differs slightly from that of Contained::Description, as shown in "The Contained Interface" on page 395. Consider the following fragment of the IDL interface for Container:

```
//IDL
interface Container : IRObject {
   // Incomplete listing of interface
   ...
   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 objects);
        ...
     };
```

`Container::Description` includes the extra member `contained_object`.

describe_ contents()

The `Container::Description` is used by the operation `describe_contents()`. This operation effectively combines calling `contents()` on the `Container` with calling `describe()` on each of the returned objects. The parameters to `describe_contents()` are as follows:

`limit_type`	A tag of type `DefinitionKind` that can be used to limit the list of contents to certain kinds of repository objects. A value of `dk_all` lists all objects.
`exclude_inherited`	This parameter is only relevant if the `Container` is an `InterfaceDef` object. In this case, it determines whether inherited definitions are included in the contents listing. `true` indicates they are excluded, while `false` indicates they are included.
`max_returned_objects`	Specifies the maximum length of the sequence that is returned.

The `describe_contents()` operation returns a sequence of `Description` structs, one for each of the `Contained` objects found.

Interface Description Structures

The `Description` struct itself serves as a wrapper for a detailed description specific to the repository object. For example, the interface `OperationDef` inherits the `OperationDef::describe()` operation.

Associated with the OperationDef interface is the struct OperationDescription. This has the following structure:

```
struct OperationDescription {
   Identifier name;
   RepositoryId id;
   RepositoryId defined_in;
   VersionSpec version;
   TypeCode result;
   OperationMode mode;
   ContextIdSeq contexts;
   ParDescriptionSeq parameters;
   ExcDescriptionSeq exceptions;
};
```

This struct is not returned directly by the operation OperationDef::describe(). Initially, it returns a Contained::Description wrapper. The first layer includes Description::kind, which in this case equals dk_Operation. The second layer includes Description::value, which is an any. This is the substance of the Description. Inside the any there is a TypeCode _tc_OperationDescription and the value of the any is the OperationDescription structure itself.

The structure of OperationDescription is as follows:

name	The name of the operation as it appears in the definition. For example, the operation account::makeWithdrawal has the name "makeWithdrawal".
id	The id is a RepositoryId for the OperationDef object. A RepositoryId is a string that uniquely identifies an object within a repository, or globally within a set of repositories if more than one is being used.
defined_in	The member defined_in gives the RepositoryId for the parent Container of the OperationDef object.
version	The version of type VersionSpec is used to indicate the version number of an Interface Repository object. This allows the Interface Repository to distinguish two or more versions of a definition with the same name, but whose details evolve over time. The OrbixWeb Interface Repository currently does not support versioning.

`result`	The `TypeCode` of the result returned by the defined operation.
`mode`	The `mode` specifies whether the operation is normal (`OP_NORMAL`) or `oneway` (`OP_ONEWAY`).
`contexts`	The member `contexts` is of type `ContextIdSeq` which is a typedef for a sequence of strings. The sequence lists the context identifiers specified in the context clause of the operation.
`parameters`	The member `parameters` is a sequence of `ParameterDescription` structs giving details of each parameter to the operation.
`exceptions`	The member `exceptions` is a sequence of `ExceptionDescription` structures giving details of the exceptions specified in the `raises` clause of the operation.

The `OperationDescription` provides all of the information present in the original definition of the operation. The CORBA specification provides for more than one way of accessing this information. The interface `OperationDef` also defines a number of attributes allowing direct access to the members of `OperationDescription`. Frequently, it is more convenient to obtain the complete description in a single step, which is why the `OperationDescription` structure is provided.

Only those interfaces that inherit from `Contained` have an associated description structure. Of those which do inherit from `Contained`, only `EnumDef`, `UnionDef`, `AliasDef` and `StructDef` have a unique associated description structure called `TypeDescription`.

The interface `InterfaceDef` is a special case. It has an extra description structure called `FullInterfaceDescription`. This structure is provided because of the special importance of `InterfaceDef` objects. It enables a full description of the interface in one step. The description is given as the return value of the special operation `InterfaceDef::describe_interface()`. Further details are given in the *OrbixWeb Programmer's Reference*.

Type Interfaces in the Interface Repository

A number of repository interfaces are used to represent definitions of types in the Interface Repository, as follows:

- `StructDef`
- `UnionDef`
- `EnumDef`
- `AliasDef`
- `InterfaceDef`
- `PrimitiveDef`
- `StringDef`
- `SequenceDef`
- `ArrayDef`

This property is independent of, and overlaps with, the properties of containment. It is useful to represent this property by inheriting these objects from an abstract base interface called `IDLType`.

This is defined as follows:

```
// IDL
// In module CORBA.
interface IDLType : IRObject {
   readonly attribute TypeCode type;
};
```

This base interface defines a single attribute giving the `TypeCode` of the defined type. This is also useful for referring to the type interfaces collectively.

The type interfaces can be classified as either named or unnamed types.

Named Types

The named type interfaces are as follows:

- `StructDef`
- `UnionDef`

- EnumDef
- AliasDef
- InterfaceDef

For example, consider the following IDL definition:

```
// IDL
enum UD {UP, DOWN};
```

This effectively defines a new type UD which for use wherever an ordinary type might appear. It is represented by an EnumDef object. More obviously, the IDL definition

```
typedef string accountName;
```

gives rise to the new type accountName.

Both these interfaces are examples of named types. This means that their definitions give rise to a new type identifier, such as "UD" or "accountName" which can be reused throughout the IDL file.

The named types StructDef, UnionDef, EnumDef and AliasDef can be grouped together by deriving from the abstract base interface TypedefDef.

Note: It is important to note that interface TypedefDef does not directly represent an IDL typedef. The interface AliasDef, which derives from TypedefDef, is the interface representing an IDL typedef.

The abstract interface TypedefDef is defined as follows:

```
// IDL
// In module CORBA.
interface TypedefDef : Contained, IDLType {
};
```

The definition of TypedefDef is trivial and causes the four named interfaces to derive from Contained in addition to IDLType. The interfaces inherit the attribute Contained::name, which gives the name of the type, and the operation Contained::describe().

For example the definition of enum UD gives rise to an EnumDef object that has an EnumDef::name of "UD". Calling EnumDef::describe() gives access to a description of type TypeDescription. The type member of the TypeDescription gives the

TypeCode of the enum. The TypedefDef interfaces all share the same description structure TypeDescription.

The interface InterfaceDef is also a named type but it is a special case. Its inheritance is given as follows:

```
// IDL
// In module CORBA.
interface InterfaceDef : Contained, Container,
IDLType {
   ...
};
```

It has three base interfaces. Since you can use IDL object references in just the same way as any ordinary type the interface IntefaceDef inherits from IDLType. For example, the definition interface account {...} gives rise to an InterfaceDef object. This object has an InterfaceDef::name that is account, and this name can be reused as a type.

Unnamed Types

The unnamed type interfaces are as follows:

- PrimitiveDef
- StringDef
- SequenceDef
- ArrayDef

These interfaces are not strictly necessary but offer an approach to querying the types in the repository that operates in parallel to the use of TypeCodes.

There are two independent approaches to querying types in the repository. The traditional approach is to provide TypeCode attributes whenever necessary so that all the types defined in the repository can be determined. However the Interface Repository also provides a complete object-oriented approach for querying the types. Consider the following example which allows you to determine the return type of getLongAddress():

```
interface Mailer {
   sequence<string> getLongAddress();
};
```

The definition of `getLongAddress()` maps to an object of type `OperationDef` in the repository. One way of querying the return type is to call `OperationDef::result_def` which returns an object reference of type `IDLType`. You can determine the type of object returned by `result_def` by obtaining the attribute `OperationDef::def_kind` inherited from `IRObject`.

In this example, the object reference is of type `SequenceDef` corresponding to the `sequence<string>` return type. To query the returned `SequenceDef` object further, obtain the attribute `SequenceDef::element_type_def`. This returns an `IDLType` which is a `PrimitiveDef` object. This `PrimitiveDef` object, in turn, has an attribute `PrimitiveDef::kind` that has a value of `pk_string`. At this stage the return type is fully determined to be a `sequence<string>`.

The alternative approach is to obtain the `TypeCode` that retrieves the complete type information in a single step at the outset. For example, the `OperationDef` object associated with `getLongAddress()` has an attribute `OperationDef::result` that gives the `TypeCode` of `sequence<string>`.

Retrieving Information from the Interface Repository

There are three ways to retrieve information from the Interface Repository:

1. Given an object reference, you can find its corresponding `InterfaceDef` object. You can determine from this all of the details of the object's interface definition.
2. Obtain an object reference to a `Repository`, the full contents can then be navigated.
3. Given a `RepositoryId`, a reference to the corresponding object in the Interface Repository can be obtained and interrogated.

These are explained in more detail in the following three subsections.

org.omg.CORBA.Object._get_interface()

Given an object reference to any CORBA object, for example, `objVar`, you can acquire an object reference to an `InterfaceDef` object as follows:

```
import org.omg.CORBA.InterfaceDef;
InterfaceDef ifVar = objVar._get_interface();
```

The member function `_get_interface()` returns a reference to an object within the Interface Repository. See the example in "Retrieving Information from the Interface Repository" on page 405 for an illustration of how to use `_get_interface()`.

For `_get_interface()` to work correctly the program must be set up to use the Interface Repository as described in "Using the Interface Repository" on page 383.

Browsing or Listing a Repository

When you obtain a reference to a `Repository` object, you can then browse or list the contents of that repository. There are two ways to obtain such an object reference as follows:

- Using `resolve_initial_references()`
- Using `bind()`

You can call the `resolve_initial_references()` operation on the ORB (`org.omg.CORBA.ORB`), passing the string "`InterfaceRepository`" as a parameter. This returns an object reference of type `org.omg.CORBA.Object`. You can then narrow this object reference to a `org.omg.CORBA.Repository` reference.

Alternatively, you can use the OrbixWeb `bind()` function, as follows:

```
import org.omg.CORBA.Repository;
import org.omg.CORBA.RepositoryHelper;
Repository repVar =
   RepositoryHelper.bind(":IFR","");
```

The operations which enable you to browse the `Repository` are provided by the interface `org.omg.CORBA.Container`. There are four provided as follows:

- `contents()`
- `describe_contents()`
- `lookup()`
- `lookup_name()`

The last two are particularly useful as they provide a facility for searching the `Repository`. The IDL for the search operations is:

```
// IDL
// In module CORBA.
interface Container : IRObject {
```

```
        ...
        Contained lookup(in ScopedName search_name);
        ...
        ContainedSeq lookup_name(
           in Identifier search_name,
           in long levels_to_search,
           in DefinitionKind limit_type,
           in boolean exclude_inherited);
        ...
    };
```

The operation `lookup()` provides a simple search facility based on a `ScopedName`. For example, consider the case where `Container` is a `ModuleDef` object representing `finance`. Passing the string "`account::balance`" to `ModuleDef.lookup()` then retrieves a reference to an `AttributeDef` object representing `balance`. This is an example of using a relative `ScopedName`. However, `lookup()` is not restricted to searching a specific `Container`. By passing an absolute `ScopedName` as an argument it is possible to search the whole `Repository` given any `Container` as a starting point. For example, given the `InterfaceDef` for `account` you can pass the string "`::finance::bank::newAccount`" to `InterfaceDef.lookup` to find the `newAccount()` operation lying within the scope of the interface `bank`.

The operation `lookup_name()` provides a different approach to searching a `Container`. Instead of the `ScopedName` it specifies only a simple name to search for within the `Container`. Because more than one match is possible with a given simple name, the `lookup()`operation can return a sequence of `Contained` objects. The parameters to `lookup_name()` are as follows:

`search_name`	Specifies the simple name of the object to search for. The OrbixWeb implementation also allows the use of "`*`" which matches any simple name.
`levels_to_search`	Specifies the number of levels of nesting to be included in the search. If set to `1`, the search is restricted to the current object. If set to `-1`, the search is unrestricted.

`limit_type`	Limits the objects which are returned. If it is set to `dk_all`, all objects are returned. If set to the `DefinitionKind` for a particular Interface Repository kind, only objects of that kind are returned. For example, if operations are of interest, you can set `limit_type` to `dk_operation`.
`exclude_inherited`	If set to `true`, inherited objects are not returned. If set to `false`, all objects, including those inherited, are returned.

Note: You cannot use `lookup_name()` to search outside of the given `Container`.

Finding an Object Using its Repository ID

You can pass a Repository ID (of type `org.omg.CORBA.RepositoryId`) as a parameter to the `lookup_id()` operation of an object reference for a repository (of type `org.omg.CORBA.Repository`). This returns a reference to an object of type `Contained`, which you can narrow to the correct object reference type.

Using the Interface Repository with the Dynamic Invocation Interface

When the Interface Repository is used in conjunction with the Dynamic Invocation Interface (DII) it is frequently necessary to retrieve type information for the parameters of an operation.

The function `org.omg.CORBA.ORB.create_operation_list()` is a convenient function that obtains the types of all the parameters in a single step. Refer to the API Reference in the *OrbixWeb Programmer's Reference* for more details.

Example of Using the Interface Repository

This section presents some sample code that uses the Interface Repository.

The following code prints the list of operation names and attribute names defined on the interface of a given object:

```
import org.omg.CORBA.*;
import org.omg.CORBA.ORB;
import org.omg.CORBA.InterfaceDefPackage.*;
try {
   //
   // Bind to the Interface Repository server
   //
   Repository ifr_repository = RepositoryHelper.bind( ":IFR" );
   //
   // Get the interface definition
   //
   Contained contained = ifr_repository.lookup( "grid" );
   InterfaceDef interfaceDef =
         InterfaceDefHelper.narrow ( contained );
   // Get a full interface description
   FullInterfaceDescription description =
         interfaceDef.describe_interface();
   // Now print out the operation names:
   System.out.println "The operation names are: ";
   for( int i = 0; i < description.operations.length; i++ )
      System.out.println( "-> " + description.operations[i].name
);
   // Now print out the attribute names:
   System.out.println "The attribute names are: ";
   for( int i = 0; i < description.attributes.length; i++ )
      System.out.println( "-> " + description.attributes[i].name
);
}
catch ( SystemException ex ){
   // Handle exceptions
}
```

You can extend the example by finding the `OperationDef` object for an operation called `doit`. You can use the `Container.lookup_name()` as follows:

```
Contained[] opSeq = null;
OperationDef opRef = null;
```

```
try
{
   interfaceDef.lookup_name
      ( "doit", 1, DefinitionKind.dk_Operation, 0 );
   if( opSeq.length != 1 ){
      System.out.println
         ( "Incorrect lookup name for lookup_name() " );
      System.exit(1);
   }
   //
   // Narrow the result to be an OperationDef
   //
   opRef = OperationDefHelper.narrow( opSeq[0] );
   .......
}
catch ( SystemException ex )
{
   // Handle Exceptions
   }
```

Repository IDs

Each Interface Repository object describing 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 identifying 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. The formats defined by CORBA are described as follows.

OMG IDL Format

This format is derived from the IDL definition's scoped name. It contains three components which are separated by colons (':') as follows:

```
IDL:<identifier/identifier/identifier/
...>:<version number>
```

The first component identifies the Repository ID format as the OMG IDL format.

The second component consists of a list of identifiers. These identifiers are derived from the scoped name by substituting "/" instead of "::".

The third component contains a version number of the format:

```
<major>.<minor>
```

Consider the following IDL definitions:

```
// IDL
interface account {
   attribute float balance;
   void makeLodgement(in float amount);
};
```

An IDL format Repository ID for the attribute `account::balance` based on these definitions is:

```
IDL:account/balance:1.0
```

This is the format of the Repository ID used by default in OrbixWeb.

DCE UUID Format

The DCE UUID format is:

```
DCE:<UUID>:<minor version number>
```

LOCAL Format

Local format IDs are intended to be used locally within an Interface Repository and are not intended to be known outside that repository. They have the format:

```
LOCAL:<ID>
```

Local format Repository IDs can be useful in a development environment as a way to avoid conflicts with Repository IDs using other formats.

Pragma Directives

You can control Repository IDs using pragma directives in an IDL source file. These pragmas allow you control over the format of a Repository ID for IDL definitions.

At present OrbixWeb supports the use of a pragma that allows you to set the version number of the Repository ID. In the present implementation of the Interface Repository you should only use one version number per Interface Repository.

Version Pragma

You can specify a version number for an IDL definition Repository ID (IDL format) using a version pragma. The version pragma directive takes the format:

```
#pragma version <name> <major>.<minor>
```

The `<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 you do not specify a version pragma for an IDL definition, the version number defaults to `1.0`. Thus the following definitions:

```
// IDL
module finance {
   #pragma version account 2.5
   interface account {
      ...
   };
};
```

yield the following Repository IDs:

```
IDL:finance:1.0
and
IDL:finance/account:2.5
```

It is important to realise that `#pragma version` does not only affect Repository IDs. If `#pragma` is used to set the version of an interface, the version number is also embedded in the stringified object reference. A client must bind to a server object whose interface has a matching version number. If the IDL interface on the server side has no version, `bind()` does not require matching versions.

Part VI

Advanced OrbixWeb Programming

22

Filters

OrbixWeb allows you to specify that additional code be executed before or after the normal code of an operation or attribute. This support is provided by allowing applications to create filters, which can perform security checks, provide debugging traps or information, maintain an audit trail, and so on.

There are two forms of filters in OrbixWeb:

- *Per-process filters.*
- *Per-object filters.*

Per-process filters monitor all operation and attribute calls leaving or entering a client's or server's address space, irrespective of the target object. Per-object filters apply to individual objects. Both of these filter types are illustrated in Figure 34 on page 416. This chapter briefly introduces each filter type, and then describes each in detail.

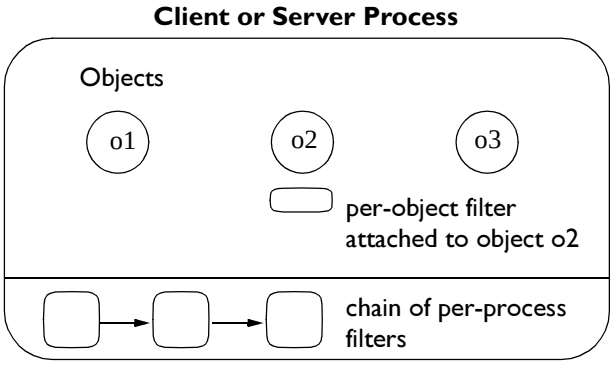

Figure 34: *Per-Process and Per-Object Filtering*

Multiple ORB Support

All parameterized calls to `ORB.init()` create a separate ORB. Each newly-created ORB instance is completely independent; for example, in terms of its configuration and listener ports. OrbixWeb allows you to associate filters with a particular ORB instance.

By default, OrbixWeb associates filters with the first fully-functional ORB created in a process. To associate a filter with a particular ORB instance, use the following constructor for your derived class:

```
protected Filter(org.omg.CORBA.ORB orb, boolean installme);
```

Refer to the *OrbixWeb Programmer's Reference* for details of `org.omg.CORBA.ORB.init()` and class `IE.Iona.OrbixWeb.Features.Filter`.

OrbixWeb also provides constructors that associate a `ThreadFilter` or an `AuthenticationFilter` with a particular ORB instance. Refer to package `IE.Iona.OrbixWeb.Features` in the *OrbixWeb Programmer's Reference* for more details.

Introduction to Per-Process Filters

Per-process filters monitor all incoming and outgoing operation and attribute requests to and from an address space. Each process can have a chain of such filters, with each element of the chain performing its own actions. You can add a new element to the chain by performing the following two steps:

- Define a class that inherits from class `Filter` (defined in package `IE.Iona.OrbixWeb.Features`).
- Create a single instance of the new class.

Pre-Marshalling Filter Points

Each filter of the chain can monitor *ten* individual points during the transmission and reception of an operation or attribute request, as shown in Figure 35 on page 419. The four most commonly-used filter points are:

- `outRequestPreMarshal` (in the caller's address space).

  This filter monitors the point prior to the transmission of an operation or attribute request from the filter's address space to any object in another address space. Specifically, it monitors the point before the operation's parameters are added to the request packet.

- `inRequestPreMarshal` (in the target object's address space).

  This filter monitors the point after an operation or attribute request has arrived at the filter's address space, but before it has been processed. Specifically, it monitors the point before the operation has been sent to the target object and before the operation's parameters have been removed from the request packet.

- `outReplyPreMarshal` (in the target object's address space).

  This filter monitors the point after the operation or attribute request has been processed by the target object, but before the result has been transmitted to the caller's address space. Specifically, it monitors the point before an operation's `out` parameters and return value have been added to the reply packet.

- `inReplyPreMarshal` (in the caller's address space).

  This filter monitors the point after the result of an operation or attribute request has arrived at the filter's address space, but before the result has been processed. Specifically, it monitors the point before an operation's `out` parameters and return value have been removed from the reply packet.

Post-Marshalling Filter Points

These four monitor points are as follows:

- `outRequestPostMarshal` (in the caller's address space).

  This filter operates the same way as `outRequestPreMarshal`, but after the operation's parameters have been added to the request packet.

- `inRequestPostMarshal` (in the target object's address space).

  This filter operates the same way as `inRequestPreMarshal`, but after the operation's parameters have been removed from the request packet.

- `outReplyPostMarshal` (in the target object's address space).

  This filter operates the same way as `outReplyPreMarshal`, but after the operation's `out` parameters and return value have been added to the reply packet.

- `inReplyPostMarshal` (in the caller's address space).

  This filter operates the same way as `inReplyPreMarshal`, but after the operation's `out` parameters and return value have been removed from the reply packet.

Failure Points

Two additional monitor points deal with exceptional conditions:

- `outReplyFailure` (in the target object's address space).

  This filter is called if the target object raises an exception, or if any preceding filter point ('in request' or 'out reply') raises an exception or uses its return value to indicate that the call should not be processed any further.

- `inReplyFailure` (in the caller's address space).

  This filter is called if the target object raises an exception or if any preceding filter point ('out request', 'in request', 'out reply' or 'in reply') raises an exception, or uses its return value to indicate that the call should not be processed any further.

Once an exception is raised or a filter point uses its return value to indicate that the call should not be processed further, no further monitor points are called (with the exception of the two failure monitor points). If this occurs in the caller's address space, `InReplyFailure` is called. If it occurs in the target object's address space, `outReplyFailure` and `inReplyFailure` are both called.

All per-process monitor points (eight marshalling points and two failure points) are shown in Figure 35 on page 419.

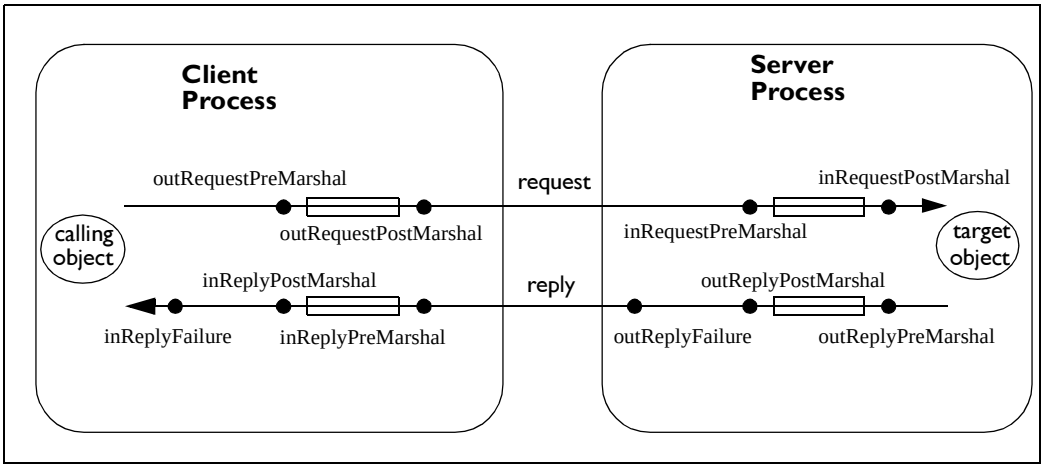

Figure 35: *Per-Process Monitor Points*

A particular filter on the per-process filter chain may perform actions for any number of these filter points, although it is common to handle four filter points, for example:

- `outRequestPreMarshal`
- `inRequestPreMarshal`
- `outReplyPreMarshal`
- `inReplyPreMarshal`

Along with monitoring incoming and outgoing requests, a filter on the client side and a filter on the server side can cooperate to pass data between them, in addition to the normal parameters of an operation (or attribute) or call. For example, you can use the 'out' filter points of a filter in the client to insert extra data into the request package; for example, using `outRequestPreMarshal`. You can use the 'in' filter points of a filter in the server to extract this data, for example, using `inRequestPreMarshal`.

Each filter point must indicate how the handling of the request should be continued once the filter point itself has completed. Specifically, a filter point can determine whether or not OrbixWeb should continue to process the request or return an exception to the caller.

Note: Per-process filters are not informed of calls between collocated objects. This is because the filters are applied only when a call leaves or arrives at an address space.

You can use a special form of per-process filter to pass authentication information from a client to a server. This type of filter is called an *authentication filter*. This supports the verification of the identity of a caller, a fundamental requirement for security. Refer to "Defining an Authentication Filter" on page 432 for more details.

Introduction to Per-Object Filters

Per-object filters are associated with a *particular* object, and not with *all* objects in an address space as in per-process filtering. Unlike per-process filters, per-object filters apply to intra-process operation requests. The following filtering points are supported:

- *Per-object pre*

  This filter applies to operation invocations on a particular object—before they are passed to the target object.

- *Per-object post*

  This filter is applied to operation invocations on a particular object—after they have been processed by the target object.

A per-object pre-filter can indicate, by raising an exception, that the actual operation call should not be passed to the target object.

To create per-object filters, perform the following steps:

1. Derive a new class from the IDL generated `Operations` class. For example, inherit from class `_GridOperations` for an object implementing interface `Grid`.
2. Create an instance of this new class. This instance behaves as a per-object filter when installed.
3. Install this filter object as either a pre-filter or as a post-filter to a particular target object.

It is important to realise that a per-object filter is either a pre-filter or a post-filter. In contrast, a single per-process filter can perform actions for any or all of its eight monitor points.

Note: You can only use per-object filtering if it was enabled when the corresponding IDL interface was compiled by the IDL compiler. Refer to "IDL Compiler Switch to Enable Object Filtering" on page 435.

The parameters to an IDL operation request are readily available for both pre and post per-object filters. Any `in` and `inout` parameters are valid for *pre* filters; `in`, `out` and `inout` parameters and return values are valid for *post* filters. In contrast, for per-process filters, parameters to the operation request are not available in general.

The per-process `inRequestPreMarshal` and `inRequestPostMarshal` filters are applied before any per-object pre-filter. The per-object post-filters are applied before any per-process `outReplyPreMarshal` and `outReplyPostMarshal` filters.

Using Per-Process Filters

To install a per-process filter, define a class deriving from the `IE.Iona.OrbixWeb.Features.Filter` class, and redefine one or more of its methods:

`outRequestPreMarshal()`	Operates in the caller's address space before outgoing requests (before marshalling).
`outRequestPostMarshal()`	Operates in the caller's address space before outgoing requests (after marshalling).
`inRequestPreMarshal()`	Operates in the receiver's address space before incoming requests (before marshalling).
`inRequestPostMarshal()`	Operates in the receiver's address space before incoming requests (after marshalling).
`outReplyPreMarshal()`	Operates in the receiver's address space before outgoing replies (before marshalling).
`outReplyPostMarshal()`	Operates in the receiver's address space before outgoing replies (after marshalling).
`inReplyPreMarshal()`	Operates in the caller's address space before incoming replies (before marshalling).
`inReplyPostMarshal()`	Operates in the caller's address space before incoming replies (after marshalling).
`outReplyFailure()`	Operates in the receiver's address space if a preceding filter point raises an exception or indicates that the call should not be processed further or if the target object raises an exception.
`inReplyFailure()`	Operates in the caller's address space if the target object raises an exception or a preceding filter point raises an exception or indicates that the call should not be processed further.

Each of the eight marshalling methods take a single parameter. This is the request on which the filtering is to take place. The return value is `boolean`, indicating whether or not OrbixWeb should continue to make the request. For example:

```
public boolean outRequestPreMarshal(org.omg.CORBA.Request r)
```

Both failure methods take two parameters: the request on which the filtering was to take place, and the exception which representing the failure of that request. The failure methods have a `void` return type. Refer to the API Reference in the *OrbixWeb Programmer's Reference* for full details of these methods.

You can obtain the details of the request being made by calling methods on the `Request` parameter. See "An Example Per-Process Filter" on page 424for more details.

The constructor of class `Filter` adds the newly created filter object into the per-process filter chain. You cannot create direct instances of `Filter`; its constructor is `protected` to enforce this. Classes derived from `Filter` normally have `public` constructors.

The marshalling methods return a value which indicates how the call should continue. Redefinitions of these methods in a derived class should retain the same semantics for the return value as specified in the relevant entries in the *OrbixWeb Programmer's Reference*.

You should define derived classes of `Filter` and redefine some subset of the filter point methods to perform the required filtering. If you do not redefine any of the non-failure monitoring methods in a derived class of `Filter`, the following implementation is inherited in all cases:

```
// Java
{ return true; } // Continue the call.
```

The failure filter methods inherit the following implementation:

```
// Java
{ return; }
```

An Example Per-Process Filter

Consider the following simple example of a per-process filter:

```
// Java

import IE.Iona.OrbixWeb.Features.Filter;
import org.omg.CORBA.Request;
import org.omg.CORBA.ORB;
import org.omg.CORBA.SystemException;

ORB orb = ORB.init(args, null);

public class ProcessFilter extends Filter {
   public boolean outRequestPreMarshal (Request r) {
      String s, o;
      try {
         s = orb.object_to_string((r.target ());
         o = r.operation ();
      }
      catch (SystemException se) {
         ...
      }
      System.out.println ("Request outgoing to "+ s
                          + " with operation name "+ o + ".");
      return true; // continue the call
   }

   boolean inRequestPreMarshal (Request r) {
      String s, o;
      try {
         s = orb.object_to_string(r.target ());
         o = r.operation ();
      }
      catch (SystemException se) {
         ...
      }
      System.out.println ("Request incoming to "+ s
                          + " with operation name " + o + ".");
      return true; // continue the call
   }
```

```
boolean outReplyPreMarshal (Request r) {
   String o;
   try {
      o = r.operation ();
   }
   catch (SystemException se) {
      ...
   }
   System.out.println ("Incoming operation "
                          + o + " finished.");
   return true; // Continue the call.
}

boolean inReplyPreMarshal (Request r) {
   String o;
   try {
      o = r.operation ();
   }
   catch (SystemException se) {
      ...
   }
   System.out.println ("Outgoing operation "
                          + o + " finished.");
   return true; // Continue the call.
}

void outReplyFailure (Request r, Exception ex) {
   String o;
   try {
      o = r.operation ();
   }
   catch (SystemException se) {
      ...
   }
   System.out.println ("Operation " + o
                          + " raised exception.");
   return;
}
```

```
   void inReplyFailure (Request r, Exception ex) {
      String o;
      try {
         o = r.operation ();
      }
      catch (SystemException se) {
         ...
      }
      System.out.println ("Operation " + o
                          + " raised exception.");
      return;
   }
}
```

Filter classes can have any name, however they *must* inherit from the class `Filter`. This class has a protected default constructor. In the example, `ProcessFilter` is given a parameterless `public` constructor by Java.

Each filter object method can examine the `Request` object it receives by calling its member functions. However, this examination must be performed in a non-destructive manner. Modification of the `Request` instance is only permitted if it is to "unwind" modifications made by a corresponding filter at the other end of the connection. This process is known as *piggybacking*. Refer to "Piggybacking Extra Data to the Request Buffer" on page 429 for more details. Modification of data inserted by the OrbixWeb runtime into the Request instance invariably causes the request to fail after the filtering stage.

Getting Additional Information about Requests

You can obtain additional information about the request by using the filter methods.

For example, you can obtain an instance of `IE.Iona.OrbixWeb.CORBA.OrbCurrent` by including the following code:

```
import IE.Iona.OrbixWeb.CORBA.OrbCurrent;
import IE.Iona.OrbixWeb._OrbixWeb;
import IE.Iona.OrbixWeb._CORBA;
....
Current curr = _OrbixWeb.ORB(orb).get_current();
OrbCurrent orbcurr = _OrbixWeb.Current(curr);
```

You can then call the `OrbCurrent()` methods on the current instance.
Refer to the description of `OrbCurrent()` in the API Reference of the *OrbixWeb Programmer's Reference*.

The following methods are of particular interest:

- `get_principal()`
- `get_object()`
- `get_server()`

Installing a Per-Process Filter

To install this per-process filter, you need only create an instance of it:

```
// Java
ProcessFilter myFilter = new ProcessFilter ();
```

This object must be created after the call to `ORB.init()` and before the handling of requests.

How to Create a System Exception

Any of the per-process filter points can raise an exception in the normal manner. Exceptions have three constructors, as shown in the following example, which uses the `NO_PERMISSION` exception:

```
public NO_PERMISSION(String reason, int minor,
                     CompletionStatus completed)
public NO_PERMISSION(int minor, CompletionStatus completed)
public NO_PERMISSION(String reason)
```

The `reason` parameter represents an exception message in text form. When using IIOP, the marshalling of this string back to a client is not supported. This is because IIOP does not permit exception `reason` strings to be passed over the wire. The client receives, instead, the string "`unknown`". The string can be marshalled successfully back to the client when using the OrbixWeb Protocol.

The `minor` parameter represents an error code used to look up an error message when reconstructing the exception on the client side.

The `completed` parameter indicates whether the requested operation succeeded. Its possible values are `COMPLETED_YES`, `COMPLETED_NO` and `COMPLETED_MAYBE`. Refer to the description of `CompletionStatus` in the API Reference of the *OrbixWeb Programmer's Reference*.

Rules for Raising an Exception

The following rules apply when a filter point raises an exception:

- Per-process filters can raise only system exceptions. Any such exception is propagated by OrbixWeb back to the caller. However, raising an exception in an `inReplyPostMarshal()` filter point does not cause the exception to be propagated. At that stage, the call is essentially already completed, and it is too late to raise an exception.
- If any filter point raises an exception, no further filter points are processed for that call, except for one or both of the failure filter points, `outReplyFailure()` and `inReplyFailure()`.
- If one of the filter points
  - `outRequestPreMarshal()`
  - `outRequestPostMarshal()`
  - `inRequestPreMarshal()`
  - `inRequestPostMarshal()`

  raises an exception, the actual operation call is not forwarded to the target application object.
- If the operation implementation raises a *user exception*, and one of the filter points
  - `outReplyFailure()`
  - `inReplyFailure()`

  raises a system exception, the system exception is raised in the calling client. The user exception is overwritten.
- If the operation implementation raises a *system exception*, no further filter points, except one or both of `outReplyFailure()` and `inReplyFailure()` are called for this invocation.

Piggybacking Extra Data to the Request Buffer

One of the `outRequest` filter points in a client can add extra piggybacked data to an outgoing request buffer. This data is then made available to the corresponding `inRequest` filter point on the server side. In addition, one of the 'out reply' marshalling filter points on a server can add data to an outgoing reply. This data is then made available to the corresponding `inReply` filter point on the client-side.

At each of the four 'out' marshalling monitor points, you can insert data by using an appropriate `org.omg.CORBA.portable.OutputStream` method for the `Request` parameter, for example:

```
// Java
import IE.Iona.OrbixWeb._OrbixWeb;
import org.omg.CORBA.Request;
import org.omg.CORBA.portable.OutputStream;
...
int l = 27;
...
try {
   OutputStream s =
      _OrbixWeb.Request(r).create_output_stream();
   s.write_long (l);
}
catch (SystemException se) {
   ...
}
```

You can extract data at each of the 'in' marshalling monitor points, using an appropriate `org.omg.CORBA.portable.Inputstream` method, for example:

```
// Java
import IE.Iona.OrbixWeb._OrbixWeb;
import org.omg.CORBA.Request;
import org.omg.CORBA.portable.InputStream;
...
int j;
...
try {
   InputStream =
      _OrbixWeb.Request(r).create_input_stream();
   j = s.read_long ();
```

```
}
catch (SystemException se) {
   ...
```

Matching Insertion and Extraction Points

You must ensure that the insertion and extraction points match correctly, as follows:

Insertion Point	Extraction Point
`outRequestPreMarshal()`	`inRequestPreMarshal()`
`outReplyPreMarshal()`	`inReplyPreMarshal()`
`outRequestPostMarshal()`	`inRequestPostMarshal()`
`outReplyPostMarshal()`	`inReplyPostMarshal()`

For example, a value inserted by `outRequestPreMarshal()` must be extracted by `inRequestPreMarshal()`. Unmatched insertions and extractions corrupt the request buffer and can cause a program crash.

When only one filter is being used, its `outRequestPostMarshal()` method can insert piggybacked data that is not removed by the corresponding `inRequestPostMarshal()` method on the called side. However, this causes problems if more than one filter is being used.

Ensuring that Unexpected Extra Data is not Passed

When coding a filter that adds extra data to the request, you should ensure that you are communicating with a server that is expecting the extra data. Frequently, a filter should add extra data only if the target object is in one of an expected set of servers.

For example,

```
outRequestPreMarshal()
outRequestPostMarshal()
inRequestPreMarshal()
inRequestPostMarshal()
```

should include the following code:

```
// Java
// First find the server name:
```

```
import org.omg.CORBA.SystemException;

String impl;

try {
1     impl = (r.target())._get_implementation().toString();
}
catch (SystemException se) {
   ...
}

if (impl.equals ("some_server")) {
   // Can add extra data.
}
else {
   // Do not add any extra data.
}
```

1. It is assumed here that the `Request` parameter is `r`.

The method `org.omg.CORBA.Object._get_implementation().toString()` returns the server name of an object reference. In this case, it returns the name of the target object.

You should not add extra data when communicating with the OrbixWeb daemon. The OrbixWeb classes may communicate with the daemon process, and you must ensure that you do not pass extra data to the daemon.

Retrieving the Size of a Request Buffer

Sometimes when programming filters you may wish to obtain the size of a `Request`; for example, in order to display trace information about traffic between OrbixWeb applications. You can obtain this information by invoking the method `getMessageLength()` on the `org.omg.CORBA.Request` class as follows:

```
// Java
import IE.Iona.OrbixWeb._OrbixWeb;
import org.omg.CORBA.SystemException;
...
int msgLen;
try {
   msgLen =
      _OrbixWeb.Request(r)[1].getMessageLength();
```

```
}
catch (SystemException se) {...}
```

Defining an Authentication Filter

Verification of the identity of the caller of an operation is a fundamental component of a protection system. OrbixWeb supports this by installing an authentication filter in every process's filter chain. This default implementation transmits the name of the principal (user name) to the server when the channel between the client and the server is first established by `bind()`. This name is also added to all requests at the server side. A server object can obtain the user name of the caller by calling the method:

```
// Java
import IE.Iona.OrbixWeb._CORBA;
...
String name = _CORBA.Orbix.get_principal_string();
```

You can override the default authentication filter by declaring a derived class of `AuthenticationFilter` and creating an instance of this class. For example, an alternative authentication filter could use a ticket-based authentication system rather than passing the caller's user name.

On the client side, a derived `AuthenticationFilter` class should override the `outRequestPreMarshal()` filter point. If this filter point alters the default behaviour, the server-side authentication filter point `inRequestPreMarshal()` must be appropriately overridden in all servers with which the client communicates.

1. For further details, see the description of `_OrbixWeb.Request()` in the API Reference in the *OrbixWeb Programmer's Reference*.

Using Per-Object Filters

You can attach a pre and/or a post per-object filter to an individual object of a given IDL type. Consider the following IDL interface:

```
// IDL
interface Inc {
   unsigned long increment(in unsigned long vin);
};
```

You can implement this as follows:

```
// Java
public class IncImplementation
   implements _IncOperations {
   public int increment (int vin) {
      return (vin+1);
   }
}
```

For example, if you have two objects of this type created, as follows:

```
// Java
Inc i1, i2;
try {
   i1 = new _tie_Inc (new IncImplementation ());
   i2 = new _tie_Inc (new IncImplementation ());
}
catch (org.omg.CORBA.SystemException se) {
   ...
}
```

you may wish to pre and/or post filter the specific object referenced by `i1`. To achieve this, define one or more additional classes that implement the `_<Interface>Operations` Java interface.

To perform pre-filtering, you can define a class, for example `FilterPre`, to have the methods and parameters specified in the `_IncOperations` Java interface:

```
// Java
public class FilterPre
   implements _IncOperations {
   public int increment (int vin) {
      System.out.println
         ("*** PRE call with parameter " + vin);
```

```
      return 0; // Here any value will do.
   }
```

Similarly, to perform post-filtering, you could define a class called `FilterPost`, as follows:

```
// Java
public class FilterPost
   implements _IncOperations {
      public int increment (int vin) {
         System.out.println
         ("*** POST call with parameter " + vin);
         return 0; // Here any value will do.
      }
}
```

In these examples, a per-object filter cannot access the object it is filtering. A filter can however access the object it is filtering by having a member variable that points to the object. You can set up this member using a constructor parameter for the filter.

To apply filters to a specific object, do the following:

```
// Java
// Create two filter objects.
Inc.Ref serverPre, serverPost;

try {
   serverPre = new FilterPre ();
   serverPost = new FilterPost ();

// Attach the two filter objects to
// the target object pointed to by i1.

((_incSkeleton)i1).__preObject = serverPre;
((_incSkeleton)il).__postObject = serverPost;
```

It is not always necessary to attach both a pre and a post filter to an object.

Attaching a pre filter to an object which already has a pre filter causes the old filter to be removed and the new one to be attached. The same applies to a post filter.

If a per-object pre filter raises an exception in the normal way, the actual operation call is not made. Normally this exception is returned to the client to indicate the outcome of the call. However, if the pre filter raises the exception `FILTER_SUPPRESS`, no exception is

returned to the caller. The caller cannot tell that the operation call has not been processed as normal.

You can raise a FILTER_SUPPRESS exception as follows:

```
// Java
import IE.Iona.OrbixWeb.Features.FILTER_SUPPRESS;
import org.omg.CORBA.CompletionStatus;
...

throw new FILTER_SUPPRESS(0, CompletionStatus.COMPLETED_NO);
```

In this example, you could use the same filter objects (those pointed to by serverPre and serverPost) to filter call to many objects. Other filters, for example a filter holding a pointer to the object it is filtering, can only be used to filter one object.

IDL Compiler Switch to Enable Object Filtering

You can apply per-object filtering to an IDL interface only if it has been compiled with the -F switch to the IDL compiler. By default, -F is not set, so object level filtering is not enabled.

Thread Filters

The class `ThreadFilter` (in package `IE.Iona.OrbixWeb.Features`) is a special kind of filter that can be used to implement custom threading and queueing policies.

This section explains the benefits of multi-threaded clients and servers, and describes class `ThreadFilter` as a mechanism for implementing multi-threaded programming with OrbixWeb.

Multi-Threaded Clients and Servers

Normally, OrbixWeb client and server programs contain one thread that starts executing at the beginning of the program (`main()`) and continues until the program terminates. Many modern operating systems enable you to create lightweight threads, with each thread having its own set of CPU registers and stack. Each thread is independently scheduled by the operating system, so it can run in parallel with the other threads in its process. The mechanisms for creating and controlling threads differ between operating systems but the underlying concepts are common.

Both clients and servers may benefit from multi-threading. However, the advantages of multi-threading are most apparent for servers.

Multi-Threaded Servers

Many servers accept one request at a time and process each request to completion before accepting the next. Where parallelism is not required, there is no need to make a server multi-threaded. However, some servers can provide improved service to their clients by processing a number of requests in parallel. Parallelism of requests may be possible because a set of clients can concurrently use different objects in the same server. Also some objects in the server can be used concurrently by a number of clients.

Benefits of Threading

Some operations can take a significant amount of time to execute. This can be because they are compute bound, or perform a large number of I/O operations, or make invocations on remote objects. If a server can execute only one such operation at a time, clients suffer because of long delays before their requests

can be started. Multi-threading enables a reduction in latency of requests, and an increase in the number of requests that a server can handle over a given period. Multi-threading also allows advantage to be taken of multi-processor machines.

The simplest threading model involves *automatically* creating a thread for each incoming request. Each thread executes the code for each call, executes the low level code that sends the reply to the caller, and then terminates. Any number of such threads can be running concurrently in a server. These can use normal synchronisation techniques, such as mutex or semaphore variables, to prevent corruption of the server's data. This protection must be programmed at two levels. The underlying ORB library must be thread safe so that concurrent threads do not corrupt internal variables and tables. Also, the application level must be made thread safe by the application programmer.

Drawbacks of Threading

The main drawbacks associated with threads are as follows:

- It may be more efficient to avoid creating a thread to execute a very simple operation. The overhead of creating a thread may be greater than the potential benefit of parallelism.
- You must ensure that application code is thread safe.

Nevertheless, multi-threaded servers are considered essential for many applications. A benefit of using OrbixWeb is that the creation of threads in a server is simple.

Threads can also be created explicitly in servers, using the threading facilities of the underlying operating system. This can be done so that a remote call can be made without blocking the server. Threads can also be created within the code that implements an operation or attribute, so that a complex algorithm can be parallelized and performed by a number of threads. These threads can be in addition to those created implicitly to handle each request.

Multi-Threaded Clients

Multi-threaded clients can also be useful. A client can create a thread and have it make a remote operation call, rather than making that remote call directly. The result is that the thread that makes the call blocks until the operation call has completed, while the rest of the client can continue in parallel. Another advantage of a multi-threaded client is that it can receive incoming operation requests to its objects without having to poll for events.

Clients must create threads explicitly, using the threading facilities of the underlying operating system. Naturally, multi-threaded clients must also be coded to ensure that they are thread safe, using a synchronisation mechanism. As for servers, the difficulty of doing this depends on the complexity of the data, the complexity of the concurrency control rules, and the form of concurrency control mechanism being used.

Thread Programming in OrbixWeb

OrbixWeb supports multi-threaded Java servers that handle multiple client requests. The Java language is multi-threaded and the OrbixWeb runtime is thread-safe.

Class ThreadFilter

The class `IE.Iona.OrbixWeb.Features.ThreadFilter` enables the implementation of custom threading and queuing policies in OrbixWeb.

The class `ThreadFilter` inherits from the class `Filter`. Although `ThreadFilter` does not redefine any of the method in the class `Filter`, it does change the behaviour of `inRequestPreMarshal()` and that of the default constructor.

To use the special functionality associated with class `ThreadFilter`, you should define a derived class of `ThreadFilter` and redefine the `inRequestPreMarshal()` method. When a request enters this filter point you can control the dispatching of the request. You can then pass the request into a custom event queue serviced by one or more threads, or you can create a thread directly and pass it the `Request` object to be dispatched.

To use the special features of the `ThreadFilter` you must use its default constructor, `Threadfilter()`. This adds a newly created object onto the `ThreadFilter` chain. You can also pass an ORB instance to the constructor to add the filter to that ORB's `ThreadFilter` chain.

Refer to the *OrbixWeb Programmer's Reference* for more details on `IE.Iona.OrbixWeb.Features.ThreadFilter`.

Models of Threading

The following are the three models of thread support provided by OrbixWeb:

- *Thread per process*
- *Thread per object*
- *Pool of threads*

Thread per Process

In this model, a thread is created for each request. Each thread executes the code for each call, executes the low level code that sends the reply to the caller, and then terminates. Any number of such threads can be running concurrently in a server.

Thread per Object

In this model, a thread is created for each object (or for a subset of the objects in the server). Each of these threads accept requests for one object only, and ignores all others. This can be an important model in real-time processing, where the threads associated with some objects need to be given higher priorities that those associated with others.

Pool of Threads

In this model, a pool of threads is created to handle incoming requests. The size of the pool puts some limit on the server's use of resources. In some cases this is better than the unbounded nature of the thread per request model. Each thread waits for an incoming request, and handles it before looping to repeat this sequence.

Implementing Threads in OrbixWeb

This section gives a brief description of how these models can be implemented in OrbixWeb.

Thread per Process

To implement this model, you should create a thread to handle a request.

The thread filter's `inRequestPreMarshal()` method can create a thread to handle an incoming request. You should use the underlying Java threads package to create the thread, and then use that thread to process the request.

The `inRequestPreMarshal()` method returns a `boolean` value. This method returns `true` when the request has been passed on. It returns `false` when the request is being handled by a separate thread.

Thread per Object

To implement this model, you should create a thread for each (or for a subset of) the objects in the server.

Each thread should have its own semaphore and queue of requests. Each thread should wait on its own semaphore. The `inRequestPreMarshal()` call should add the `Request` to the correct queue of requests, and signal the correct semaphore.

When the thread awakens, it should call `continueThreadDispatch()` to process the topmost request, and then loop to await the next one.

Pool of Threads

To implement this model, a pool of threads should be created, and each thread should wait on a shared semaphore.

When a request arrives, the `inRequestPreMarshal()` function of the `ThreadFilter` should place a pointer to the `Request` in an agreed variable and signal the semaphore. Alternatively, a queue can be used.

One of the threads awakens, and should call `continueThreadDispatch()` before looping to repeat the sequence.

The three models of threading are illustrated by the `Threads` demonstrations in the `demos` directory of your OrbixWeb installation.

23

Smart Proxies

Smart proxies are an OrbixWeb-specific feature that allow you to implement proxy classes manually, thereby allowing client interaction with remote services to be optimized. This chapter describes how proxy objects are generated, and the general steps needed to implement smart proxy support for a given interface. It also describes how a you can build a simple smart proxy. This example is based on a small load balancing application.

The IDL compiler automatically generates proxy classes for IDL interfaces. Proxy classes are used to support invocations on remote interfaces. When a proxy receives an invocation, it packages the invocation for transmission to the target object in another address space on the same host, or on a different host.

Proxy Classes and Smart Proxy Classes

This section describes how OrbixWeb manages proxies.

Proxy Classes

For each IDL interface, the OrbixWeb IDL compiler generates a Java interface defining the client view of the IDL interface. It also generates a Java *proxy class*, which implements proxy functionality for the methods defined in the Java interface. The proxy class gives the code for standard proxies for that IDL interface—these proxies transmit requests to their real object and return the results they receive to the caller.

Smart Proxy Classes

A smart proxy class is a user-defined alternative to the IDL-generated proxy class. OrbixWeb implicitly constructs a standard proxy when an object reference enters the client address space. Experienced Orbix developers should note that OrbixWeb does not use proxy factory classes to construct standard proxy objects. However, OrbixWeb does not implicitly create smart proxies, so each smart proxy class depends on the implementation of a corresponding class that manufactures smart proxy objects when requested to by OrbixWeb. This class is called a *smart proxy factory class*.

Requirements for Smart Proxies

To provide smart proxies for an IDL interface, do the following:

1. Define the smart proxy class, which must inherit from the generated proxy class.
2. Define a smart proxy factory class, which creates instances of the smart proxy class on request. OrbixWeb calls the proxy factory's `New()` method whenever it wishes to create a proxy for that interface.
3. Create a single instance of the proxy factory class in the client program.

Note: Apart from the introduction of new classes and the creation of the proxy factory object, no changes are required to existing clients in order to introduce smart proxy functionality. In particular, their operation invocation code remains unchanged.

Once you have performed these steps, OrbixWeb communicates with the smart proxy factory whenever it needs to create a proxy of that interface.There are three cases, as follows:

- When the interface's `bind()` method is called.
- When a reference to an object of that interface is passed back as an `out` or `inout` parameter or a return value, or when a reference to a remote object enters an address space via an `in` parameter.
- When `ORB.string_to_object()` is called with a stringified object reference for a proxy of that interface.

You can define more than one smart proxy class, and associated smart proxy factory class for a given IDL interface. OrbixWeb maintains a linear linked list of all of the proxy factories for a given IDL interface.

A chain of smart proxy factories is allowed for an IDL interface because the same IDL interface can be provided by a number of different servers in the system. It may be useful, therefore, to have different smart proxy code to handle each server, or set of servers. Each factory in turn can examine the marker and server name of the target object for which the proxy is to be created. The factory class can then decide whether to create a smart proxy for the object or to defer the request to the next proxy factory in the chain.

Creating a Smart Proxy

The following steps must be performed in order to create a smart proxy:

1. Implement the smart proxy class.

   The constructor(s) of this class are used by the proxy factory.
2. Implement a new proxy factory class, derived from the OrbixWeb `ProxyFactory` class (defined in package `IE.Iona.OrbixWeb.Features`). It should redefine the `New()` method to create new smart proxy objects. It may also return null to indicate that it is not willing to create a smart proxy.
3. Declare an object of this new class. The inherited base class constructor automatically registers this new proxy factory with the factory manager object.

When a new proxy is required, OrbixWeb calls all of the registered proxy factories for the class until one of them successfully builds a new proxy. If none succeeds, a standard proxy is implicitly constructed. Proxy factories are automatically added to the chain of factories as they are created. However, you cannot predict the order of use of smart proxy factories.

The factory manager requests each proxy factory to manufacture a new proxy using its `New()` method:

```
// Java
// The String parameter is the full object
// reference of the target object.
// The return value is the new smart proxy
// object.
import org.omg.CORBA.portable.Delegate;
...
public org.omg.CORBA.Object New (Delegate d);
```

If the `New()` method returns null, OrbixWeb tries the next smart proxy factory in the chain.

Examples of these smart proxy implementation steps are given in the rest of this chapter.

Multiple ORB Support

All parameterized calls to `ORB.init()` create a separate ORB. Each newly-created ORB instance is completely independent; for example, in terms of its configuration and listener ports. OrbixWeb allows you to associate smart proxies with particular ORB instances.

By default, OrbixWeb associates smart proxies with the first fully-functional ORB created in a process. To associate a smart proxy with a particular ORB instance, use the following constructor for your derived class:

```
protected ProxyFactory(org.omg.CORBA.ORB orb, String name);
```

The `orb` parameter associates the smart proxy with a specific ORB instance. The `name` parameter refers to name of the IDL interface implemented by the smart proxy object

Refer to the *OrbixWeb Programmer's Reference* for details of class `IE.Iona.OrbixWeb.Features.ProxyFactory` and the `org.omg.CORBA.ORB.init()` method.

Benefits of Using Smart Proxies

It is sometimes beneficial to be able to implement proxy classes manually. The circumstances in which the use of smart proxies may be advantageous include the following:

- **Load Balancing**

  For client programmers, a typical example is where you want to introduce *load balancing* between several remote objects when invoking operations. For example, if multiple remote objects can meet a request for a computationally intensive operation, a client application may wish to route each invocation to the object that is currently least busy.

- **Caching Information**

  For interface implementers, it is often useful to implement smart proxies to *cache* some information from a remote object locally at a client site. In the simple bank application you may wish, for example, to cache the balance of an account at a client. Requests to obtain the balance of the account can then be immediately satisfied, provided you ensure that withdrawals and deposits to the account refresh the cached value.

Example: A Simple Smart Proxy

Consider a very simple example of a load balancing system, based on the following IDL definition:

```
// IDL

interface NumberCruncher {
   long crunch (in long number);
};

interface NCManager {
   // Get the least loaded number cruncher:
   NumberCruncher getNumberCruncher ();
};
```

In this application, it is assumed that a number of objects exist that implement the `NumberCruncher` interface. Each of these objects is capable of exhibiting individual load characteristics; this is the case, for example, if each is located in a separate OrbixWeb server process.

It is also assumed that an OrbixWeb server exists that implements the `NCManager` interface. The `NCManager` implementation object is responsible for locating the currently least-loaded `NumberCruncher` and returning the corresponding object reference to the client. The client can then invoke the `crunch()` operation, perhaps repeatedly, on the target object.

Of course, the load on each `NumberCruncher` object changes over time. If it is valid to direct each client `crunch()` invocation to *any* `NumberCruncher` object, the performance perceived by the client can be improved by updating the target object before each operation call. In this example, a smart proxy is implemented which takes advantage of this fact to optimize the performance of the `crunch()` operation.

Creating a Smart Proxy

The following two steps are required when creating a smart proxy:

- Define a Smart Proxy Class.
- Define a Proxy Factory for Smart Proxies.

Defining a Smart Proxy Class

Define a smart proxy class, called `SmartNC`, for Java proxy class `NumberCruncher`. Instances of this class stores a variable holding a default proxy for the `NumberCruncher` object. This proxy variable is updated before each call to `crunch()`, and the operation invocation is then routed via the refreshed default proxy.

```
// Java
package SmartProxy;

import org.omg.CORBA.SystemException;

1   public class SmartNC
       extends _NumberCruncherStub {

       // Store an NCManager proxy
       private NCManager theNCManager;
```

```
2       public SmartNC () {

           // Create NCManager proxy
           try {
              theNCManager = NCManagerHelper.bind ();
           }
           catch (SystemException se) {
                 ...
           }
        }

3       public int crunch (int number) {
           NumberCruncher actNC = null;

           // Create default proxy for current
           // least busy NumberCruncher object
           try {
                 actNC = theNCManager.getNumberCruncher ();
           }
           catch (SystemException se) {
                 ...
           }

           // Make remote invocation
           return actNC.crunch (number);
        }
     }
```

1. Class `SmartNC` inherits from the default proxy class generated by the IDL compiler. It therefore inherits all of the code required to make a remote invocation: if required, each `SmartNC` method can make a call-up to its base class's method to make a remote call. However, this functionality is not required in this example.
2. The `SmartNC` constructor initialises a member variable holding a proxy for the `NCManager` object by calling `NCManagerHelper.bind()`.
3. The `crunch()` method first obtains a default proxy for the current least loaded `NumberCruncher` object by invoking `NCManager.getNumberCruncher()`. The implementation of the smart proxy factory class, described in "Defining a Proxy Factory for Smart Proxies", prevents this invocation from creating a second smart proxy.

The smart `crunch()` method then invokes the default `crunch()` on the newly created object.

Defining a Proxy Factory for Smart Proxies

Define a new proxy factory to generate the smart proxies at the appropriate time. Recall that the base class for all proxy factory classes is the following class: `IE.Iona.OrbixWeb.Features.ProxyFactory`.

```
// Java
package SmartProxy;

import IE.Iona.OrbixWeb.Features.ProxyFactory;
import org.omg.CORBA.portable.Delegate;
import org.omg.CORBA.portable.ObjectImpl;
import org.omg.CORBA.SystemException;
import org.omg.CORBA.Object;
...
1   public class SmartNCFactory
       extends ProxyFactory {

       // Flag to indicate whether a smart proxy
       // or a true proxy should be created
       private static boolean createProxy;

       public SmartNCFactory () {
          super (NumberCruncherHelper.id());
          createProxy = true;

       }

2      public Object New(Delegate d) {
          // You only need one smart proxy to
          // manage the default proxies, so
          // allow implicit creation of a default
          // proxy (if a smart proxy already exists)
          if (createProxy == false)
             return null;

          createProxy = false;
```

```
3          // Create a smart proxy
           ObjectImpl new_ref = null;
           try {
              new_ref = new SmartNC ();
              new_ref._set_delegate(d);
           }
           catch (SystemException ex) {
              return null;
           }
           return new_ref;
        }
     }
```

This code is described as follows:

1. The member initialization list of the constructor of class `SmartNCFactory` makes a call to the constructor of class `ProxyFactory`. The parameter passed is the return value of the static method `NumberCruncherHelper.id()`. This automatically generated method returns a string which holds information about the IDL interface type for the proxy.

   The proxy and proxy factory class hierarchies are shown in Figure 36.

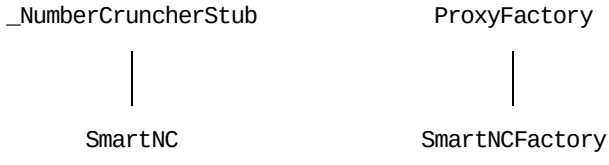

Figure 36: *Class Hierarchy for Smart Proxy Classes*

2. The `SmartNCFactory.New()` method is called by OrbixWeb to signal that a smart proxy can be created. OrbixWeb passes it an object of type `org.omg.CORBA.portable.Delegate`.

   If the method decides to create a smart proxy, it must instantiate a new smart proxy It must also set the delegate object using the `_set_delegate()` operation which all proxies inherit from `org.omg.CORBA.portable.ObjectImpl`.

3. In this example, each client only requires a single smart proxy object to manage all invocations on class `NumberCruncher`. The `New()` method first checks the member variable `createProxy` member variable to determine if it needs to create a smart proxy.

   If the value of this variable is `false`, the method simply returns `null`. This results in the invocation of the next smart proxy factory in the factory chain, or the creation of a default proxy object (if this is the last factory in the chain).

A Sample Client

Finally, you must declare a single instance of the new proxy factory class in the client:

```
// Java
SmartNCFactory ncFact = new SmartNCFactory ();
```

The inherited base class constructor then registers this new factory, and enters it into the linked list of factories for interface `NumberCruncher`.

You can code a sample client that communicates using this smart proxy as follows:

```
// Java
package SmartProxy;

import org.omg.CORBA.SystemException;

public class Client {
   static public void main (String argv[]) {
      NumberCruncher ncRef = null;
      NCManager ncmRef = null;
      SmartNCFactory ncFact =
         new SmartNCFactory ();
      int result1 = 0;
      int result2 = 0;
      int result3 = 0;

      try {
1        // bind to NCManager
         ncmRef = NCManagerHelper.bind ();

         // get least loaded number cruncher
         ncRef = ncmRef.getNumberCruncher ();
```

```
2              // do some calculations
               result1 = ncRef.crunch (100);
               result2 = ncRef.crunch (200);
               result3 = ncRef.crunch (300);
            }
            catch (SystemException se) {
               System.out.println (
                  "Number crunch failed.");
               System.out.println (se.toString ());
            }
         }
      }
```

This code can be described as follows:

1. The client binds to the `NCManager` object, from which it obtains an object reference for the currently least-loaded `NumberCruncher`. When this object reference enters the client address space, a smart proxy is created transparently to the client.
2. The client invocations on operation `crunch()` are then automatically routed through the smart proxy, as previously described in this chapter.

24

Loaders

This chapter describes the use of loaders, an OrbixWeb-specific feature designed to support persistent objects.

When an operation invocation arrives at a server process, OrbixWeb searches for the target object in the internal object table for the process. By default, if the object is not found, OrbixWeb returns an exception to the caller. However, if one or more *loader* objects are installed in the process, OrbixWeb informs the loader about the *object fault* and allows it to load the target object and resume the invocation transparently to the caller. OrbixWeb maintains the loaders in a chain, and tries each loader in turn until one can load the object. If no loader can load the object, an exception is returned to the caller.

Loaders can provide support for persistent objects—long-lived objects stored on disk in the file system or in a database.

Loaders are also called when an object reference enters an address space, and not only when a missing object is the target of a request. This can arise in a number of ways:

- When a call to either of the methods `bind()` or `string_to_object()` is made from within a process.
- For a server: as an `in` parameter.
- For a client (or a server making an operation call): as an `out` or `inout` parameter, or a return value.

The loaders can respond to such object faults by loading the target object of the reference into the process's address space. If no loader can load the referenced object, OrbixWeb constructs a proxy for the object.

Overview of Creating a Loader

To code a loader, define a derived class of `LoaderClass` (defined in package `IE.Iona.OrbixWeb.Features`). To install a loader, create an instance of that new class. `LoaderClass` provides the following methods:

- `load()`

  OrbixWeb uses this method to inform a loader of an object fault. The loader is given the marker of the missing object so that it can identify which object to load.

- `save()`

  When a process terminates, the objects in its address space can be saved by its loaders. To allow this, OrbixWeb supplies a `shutdown()` method, to call on the `_CORBA.Orbix` object before process termination. `_CORBA.Orbix.shutdown()` makes an individual call to `save()` for each object managed by a loader. You can also explicitly call the `save()` method through the `IE.Iona.OrbixWeb.CORBA.ObjectRef._save()` method. The `_OrbixWeb.Object()` cast operation must be used on any `org.omg.CORBA.Object` object before calling `_save()` because this method is on the OrbixWeb-specific `ObjectRef` interface.

- `record()` and `rename()`

  These methods are used to control naming of objects, and they are explained in Chapter 8 "Making Objects Available in OrbixWeb" on page 171.

The constructor of `LoaderClass` (the base class of all loaders) takes an optional `boolean` parameter. When creating a loader object, this parameter must be `true` if the `load()` method of the new loader is to be called by OrbixWeb.

Multiple ORB Support

All parameterized calls to `ORB.init()` create a separate ORB. Each newly-created ORB instance is completely independent; for example, in terms of its configuration and listener ports. OrbixWeb allows you to associate loaders with particular ORB instances.

By default, OrbixWeb associates loaders with the first fully-functional ORB created in a process. To associate a loader with a particular ORB instance, use the following constructor for your derived class:

```
public LoaderClass(org.omg.CORBA.ORB orb, boolean registerMe);
```

You should refer to the *OrbixWeb Programmer's Reference* for more details on class `LoaderClass`.

Refer to the section "Example Loader" on page 460 for sample code. The sections before this explain the different aspects of the loader mechanism in more detail.

Specifying a Loader for an Object

Each object has an associated a loader object. OrbixWeb informs the loader object when the object is named, renamed or saved. If an object does not have a specified loader, OrbixWeb associates it with a default loader.

You can specify an object's loader as the object is being created, either using the TIE or the ImplBase approach.

TIE Approach

Using the TIE approach, you can pass the loader object as the third parameter to a TIE object constructor. For example,

```
// Java
// myLoader is a loader object:

bank bRef = new _tie_bank
   (new bankImplementation (),
   "College Green", myLoader);
```

ImplBase Approach

Using the ImplBase approach, you can declare the implementation class's constructor to take a loader object parameter; and define this constructor to pass on this object as the second parameter to its ImplBase class's constructor. For example:

```
// Java
import org.omg.CORBA.SystemException;
import IE.Iona.OrbixWeb.Features.LoaderClass;

class bankImplementation extends _bankImplBase {
            ...
   public bankImplementation
            (String marker, LoaderClass loader) {
               super (marker, loader);
```

```
                ...
    }
}
```

OrbixWeb associates each object with a simple default loader if it does not have a specified loader. This loader does not support persistence.

You can retrieve an object's loader by calling:

```
// Java
// In package IE.Iona.OrbixWeb.CORBA
// in interface ObjectRef
import IE.Iona.OrbixWeb.Features;
...
public LoaderClass _loader ();
```

Connection between Loaders and Object Naming

When supporting persistent objects, you often need to control the markers that are assigned to them. For example, you may need to use an object's marker as a key to search for its persistent data. The format of these keys depends on how the persistence is implemented by the loader. Therefore, it is common for loaders to choose object markers. Loaders can accept or reject markers chosen by application level code.

Recall that you can name an object in a number of ways:

- By passing a marker name to a TIE object constructor, for example:

  ```
  bankRef bRef = new _tie_bank
     (new bankImplementation (), "College Green",
                                          myLoader);
  ```

- By passing the marker name to the `BOAImpl` constructor, for example:

  ```
  bankImplementation bImpl;

  try {
     bImpl = new bankImplementation
        ("College Green", myLoader);
  }
  ...
  ```

- By calling `IE.Iona.OrbixWeb.CORBA.ObjectRef._marker(String)`, for example:

```
import IE.Iona.OrbixWeb._OrbixWeb;
...
org.omg.CORBA.Object bRef = //obtained using bind
                            //or Naming Service

_OrbixWeb.Object(bRef)._marker ("Foster Place");
```

In all cases, OrbixWeb calls the object's loader to confirm the chosen name, thus allowing the loader to override the choice. In the first two cases above, OrbixWeb calls `record()`; in the last case it calls `rename()` because the object already exists.

OrbixWeb executes the following algorithm when an object is created, or an object's existing marker is changed:

- If the specified marker is not null, OrbixWeb checks if the name is already in use in the process. If it is not in use, the name is suggested to the loader (by calling `record()` or `rename()`). The loader can accept the name by not changing it. Alternatively, the loader can reject it by changing it to a new name. If the loader changes the name, OrbixWeb again checks that the new name is not already in use within the current process; if it is already in use, the object is not correctly registered.
- If no name is specified or if the specified name is already in use within the current process, OrbixWeb passes a null value to the loader (by calling `record()` or `rename()`) which must then choose a name. OrbixWeb then checks the chosen name; the object is not correctly registered if this chosen name is already in use.

Both `record()` and `rename()` can, if necessary, raise an exception.

The implementations of `rename()` and `record()` in `LoaderClass` both return without changing the suggested name. Its implementations of `load()` and `save()` perform no actions.

The default loader (associated with all objects not explicitly associated with another loader) is an instance of `NullLoaderClass`, a derived class of `LoaderClass`. This class inherits `load()`, `save()` and `rename()` from `LoaderClass`. It implements `record()` so that if no marker name is suggested it chooses a name that is a unique string of decimal digits.

Loading Objects

When an object fault occurs, the `load()` method is called on each loader in turn until one of them successfully returns the address of the object, or until they have all returned `null`.

The responsibilities of the `load()` method are:

- To determine if the required object is to be loaded by the current loader.
- If so, to re-create the object and assign the correct marker to it.

The `load()` method is given the following information:

- The interface name.
- The target object's marker.
- A `boolean` value, set as follows depending on why the object fault occurred:

  `true` Because of a call to `bind()` or `string_to_object()` by the process that contains the loader.

  `false` Because of an object fault on the target object of an incoming operation invocation, or on an `in`, `out` or `inout` parameter or return value.

You can determine the interface name of the missing object as follows:

- If an object fault occurs because of the call:

  ```
  p = I1Helper.bind( <parameters> );
  ```

  the interface name in `load()` will be "`I1`".

  If the first parameter to the `bind()` is a full object reference string, OrbixWeb returns an exception if the reference's interface field is not `I1` or a derived interface of `I1`.

- If an object fault occurs during the call

  ```
  p = _CORBA.Orbix.string_to_object
     ( <full object reference string> );
  ```

  the interface name in `load()` is that extracted from the full object reference string.

- If a loader is called because of a reference entering an address space (as an `in`, `out` or `inout` parameter, a return value, or as the target object of

an operation call), the interface name in `load()` is the interface name extracted from the object reference.

Saving Objects

You can invoke the method `_CORBA.Orbix.shutdown()` before the application exits. If this method is invoked, OrbixWeb iterates through all of the objects in its object table and calls the `save()` method on the loader associated with each object. A loader can save the object to persistent storage, either by calling a method on the object, or by accessing the object's data and writing this data itself. The `_save()` method is also called if `disconnect()` or `dispose()` is called for the object.

You can also explicitly cause the `save()` method to be called by invoking an object's `_save()` method. The `_save()` method calls the `save()` method on the object's loader. You must call the `_save()` in the same address space as the target object: calling it in a client process, on a proxy, has no effect.

The two alternative invocations of `save()` are distinguished by its second parameter. This parameter is of type `int`, and takes one of the following values:

`_CORBA.processTermination`	The process is about to exit.
`_CORBA.objectDeletion`	The method `BOA.dispose` or method `BOA.disconnect()` has been called on the object.
`_CORBA.explicitCall`	The object's `_save()` method has been called.

Writing a Loader

To write a loader for a specific interface, you normally perform the following actions:

1. Redefine the `load()` method to load the object on demand. Normally, you use the object's marker to find the object in the persistent store.
2. Redefine the `save()` method so that it saves its objects on process termination, and also when `_save()` is called.
3. Redefine the `record()` and `rename()` methods normally. Often, `record()` chooses the marker for a new object; and `rename()` is sometimes written to prevent an object's marker being changed. However, `record()` and `rename()` are sometimes not redefined in a

simple application, where the code that chooses markers at the application level can be trusted to choose correct values.

Example Loader

This section presents a simple loader for one IDL interface. A version of the code for this example is given in the `demos\loaders_per_simp` directory of your OrbixWeb installation.

There are two interfaces involved in the application:

```
// IDL
// In file bank.idl.

interface account {
   readonly attribute float balance;
   void makeLodgement(in float f);
   void makeWithdrawal(in float f);
};

interface bank {
   account newAccount(in string name);
};
```

This simple example assumes that these definitions are compiled using the IDL `-jP` switch as follows:

```
idl -jP loaders_per_simp bank.idl
```

The classes output by the IDL compiler are within the scope of the `loaders_per_simp` Java package.

Interfaces `account` and `bank` are implemented by classes `accountImplementation` and `bankImplementation`, respectively. Instances of class `accountImplementation` are made persistent using a loader (of class `Loader`). The persistence mechanism used is very primitive because it uses one file per account object. Nevertheless, the example acts as a simple introduction to loaders. The implementation of class `Loader` is shown later, but first the implementations of classes `accountImplementation` and `bankImplementation` are shown.

You can implement class accountImplementation as follows:

```
// Java

package loaders_per_simp;

import IE.Iona.OrbixWeb.Features.LoaderClass;
import org.omg.CORBA.SystemException;
import org.omg.CORBA.Object;
public class accountImplementation
   implements _accountOperations {
   protected String m_name;
   protected float m_balance;
   protected String m_accountNr;

   public accountImplementation
      (float initialBalance, String name,
                              String nr) {
      // Initialize member variable values.
      // Details omitted.
   }

   // Methods to implement IDL operations:
   public float balance () {
      return m_balance;
   }

   public void makeLodgement (float f) {
      m_balance += f;
   }

   public void makeWithdrawal (float f) {
      m_balance -= f;
   }

   // Methods for supporting persistence.
   public static Object loadMe
      (String file_name, LoaderClass loader) {
      // Details shown later.
   }
```

```
        public void saveMe (String file_name) {
           // Details shown later.
        }
     };
```

Two methods are added to the implementation class. The `load()` method of the loader calls the static method `loadMe()`. This is given the name of the file to load the account from. The method `saveMe()` writes the member variables of an account to a specified file. You can code these methods as follows:

```
public static Object loadMe
           (String file_name, LoaderClass loader) {
   ...

   RandomAccessFile file = null;
   String name = null;
   float bal = 0;

   try {
      file = new RandomAccessFile (file_name, "r");
      name = file.readLine ();
      bal = file.readFloat ();
      file.close();
   }
   catch (java.io.IOException ex) {
      ...
      System.exit (1);
   }
   accountImplementation aImpl = new
      accountImplementation (bal, name, file_name);
   account aRef = new
      _tie_account (aImpl, file_name, loader);

   return aRef;
}

public void saveMe (String file_name) {
   ...
   RandomAccessFile file = null;
```

```
   try {
      file = new RandomAccessFile (file_name, "rw");
      file.seek (0);
      file.writeBytes (m_name + "\n");
      file.writeFloat (m_balance);
      f.close();
   }
   catch (java.io.IOException ex) {
      ...
      System.exit(1);
   }
}
```

The statement:

```
account aRef = new _tie_account (aImpl, file_name, loader);
```

in `accountImplementation.loadMe()` creates a new TIE for the implementation object `accImpl`, and specifies its marker to be `file_name` and its loader to be the loader object referenced by parameter `loader`. Actually, this example creates only a single loader object as shown in the next code sample.

Class `bankImplementation` is implemented as follows:

```
// Java

package loaders_per_simp;

import IE.Iona.OrbixWeb.Features.LoaderClass;
import org.omg.CORBA.SystemException;

public class bankImplementation
   implements _bankOperations {
   protected int m_sortCode;
   protected int m_lastAc;
   protected LoaderClass m_loader;

   public bankImplementation (long sortCode,
      LoaderClass loader) {
      m_sortCode = sortCode;
      m_loader = loader;
      m_lastAc = 0; // Number of previous account.
   }
```

```
   // Method to implement IDL operation:
   public account newAccount (String name) {
      String accountNr = new String ("a"
         + m_sortCode + "-" + (++m_lastAc));

      accountImplementation aImpl = null;
      try {
         aImpl = new accountImplementation
            (100, name, accountNr);
      }
      catch (SystemException se) {
         ...
      }

      account aRef = new _tie_account(aImpl, accountNr, m_loader);
      return aRef;
   }
}
```

The main method creates a single loader object, of class `Loader`, and each `account` object created is assigned this loader. Each `bankImplementation` object holds its sort code (a unique number for each bank, for example `1234`), and also a reference to the loader object to associate with each `account` object as it is created. Each account is assigned a unique account number, constructed from its bank's sort code and a unique counter value. The first account in the bank with sort code `1234` is therefore given the number "`a1234-1`". The marker of each account is its account number, for example "`a1234-1`". This ability to choose markers is an important feature for persistence.

The statement:

```
account aRef =
   new _tie_account (aImpl, accountNr, m_loader);
```

creates a new TIE for the `accountImplementation` object assigning it the marker `accountNr` and the loader referenced by `m_loader`. (The bank objects are not associated with an application level loader, so they are implicitly associated with the OrbixWeb default loader.)

The server application class must create a loader and a bank; for example:

```
// Java
package loaders_per_simp;

import org.omg.CORBA.SystemException;

public class bankServer {
   public static void main (String args[]) {
      Loader myLoader = new Loader ();
      bankImplementation bankImpl =
         new bankImplementation (1234, myLoader);
      bank bRef;

      try {
         bRef = new _tie_bank (bankImpl, "b1234");
      }
      catch (SystemException se) {
         ...
      }
      ...
   }
}
```

Coding the Loader

You can implement class `Loader` as follows:

```
// Java
// In file Loader.java.
package loaders_per_simp;

import org.omg.CORBA.SystemException;
import org.omg.CORBA.Object;

import IE.Iona.OrbixWeb.CORBA.Features.LoaderClass;
import IE.Iona.OrbixWeb._CORBA;
import IE.Iona.OrbixWeb._OrbixWeb;

class Loader extends LoaderClass {
   public Loader() {
    super (true);
   }
```

```
   public Object load (String interfaceMarker,
      String marker, boolean isBind) {
      // There will always be an interface;
      // but the marker may be the null string.
      if (marker!=null && !marker.equals ("")
         && marker.charAt (0)=='a' &&
         interface.equals ("account"))
         return accountImplementation.loadMe
            (marker, this);
      return null;
   }

   public void save (Object obj, int reason) {
      String marker = _OrbixWeb.Object(obj)._marker ();

      if (reason == _CORBA.processTermination) {
         accountImplementation impl =
            (accountImplementation)(((_tie_account)obj)._deref());

         aImpl.saveMe (marker);
      }
   }
}
```

The constructor of `LoaderClass` takes a parameter indicating whether or not the loader being created should be included in the list of loaders tried when an object fault occurs. By default, this value is `false`; so the loader class's constructor passes a value of `true` to the `LoaderClass` constructor to indicate that instances of `Loader` should be added to this list.

The `accountImplementation.loadMe()` method assigns the correct marker to the newly created object. If it failed to do this, subsequent calls on the same object result in further object faults and calls to the `Loader.load()` method.

It is possible for the `Loader.load()` method to read the data itself, rather than calling the static method `accountImplementation. loadMe()`. However, to construct the object, `load()` dependent on there being a constructor on class `accountImplementation` that takes all of an account's state as parameters. Since this is not be the case for all classes, it is safer to introduce a method such as `loadMe()`. Equally, `Loader.save()` can access the account's data and write it out, rather than calling `accountImplementation.saveMe()`. However, it is then dependent on `accountImplementation` providing some means to access all of its state.

In any case, having `loadMe()` and `saveMe()` within class `accountImplementation` provides a sensible split of functionality between the application level class, `accountImplementation`, and the loader class.

Client-Side

Loaders are transparent to clients. A client that wishes to create a specific account could execute the following:

```
// Java

bank bRef;
account aRef;

try {
   // Find the bank somehow; for example,
   // using bind():
   bRef = bankHelper.bind (":per_simp", host);

   aRef = bRef.newAccount ("John");
}
catch (SystemException se) {
   ...
}
```

A client that wishes to manipulate an account can execute the following:

```
// Java
// To access account with account
// number "a1234-1".
account aRef;
float bal;

try {
   aRef = accountHelper.bind
      ("a1234-1:per_simp", host);
   bal = aRef.balance ();
   aRef.makeWithdrawal (100.00);
}
catch (SystemException se) {
   ...
}
```

If the target account is not already present in the server then the `load()` method of the loader object is called. If the loader recognises the object, it handles the object fault by re-creating the object from the saved data. If the load request cannot be handled by that loader, then the default loader is tried next and this always indicates that it cannot load the object. This finally results in an `org.omg.CORBA.INV_OBJREF` exception being returned to the caller.

Polymorphism

Every loader you write should allow for polymorphism. In particular, the interface name passed to a loader *may be a base interface of the actual interface* that the target object implements. This may arise, for example, when the client has bound to an object using `I1Helper.bind()` but where the object's actual interface is in fact a derived interface of `I1`.

The class of the target object must therefore be determined either from the marker passed to the loader, or from the data used to load the target object. The demonstration code for loaders shows the marker names being used to distinguish the real interface of an object, using the first character of each marker. This is a simple approach, but it is probably better in a large system to use some information stored with the persistent data of each object.

You must also remember that it may not be necessary to distinguish the real interface of an object in all applications and for all interfaces. If you always use the correct interface name in calls to `bind()` (that is, you always used `I1Helper.bind()` when binding to an object with interface `I1`) handling polymorphism is not required. This is also the case if you do not use `bind()` for a given interface: for example, you may obtain all object references to accounts by searching (say, using an owner name) in a bank, rather than using `bind()`.

It is however possible that, because of programmer error, the actual interface of the target object is not the same or a derived interface of the correct one. This should be detected by a loader.

Approaches to Providing Persistent Objects

There are many ways to use the support described so far in this chapter. This section outlines some of the choices available.

The information provided to a loader on an object fault comprises the object's *marker* and the *interface* name. The loader must be able to find the requested object using these two pieces of information. It must also be able to determine the implementation class of the target object—so that it can create an object of the correct class. Naturally, this implementation class must implement the required interface or one of its derived interfaces.

It is normal, therefore, to use the marker as a key to find the object, and either to encode the target object's implementation class in the marker, or to first find the object's persistent state and determine the implementation class from that data.

For example, a prefix of the marker could indicate the implementation class and the remainder of the marker could be the name of the file that holds the object's persistent state.

The following are some of the choices available when using loaders to support persistent objects:

- You can store each object in its own file, or you may use a record system in which one or more records represent an object. You can store records, for example, in a relational database management system, or by using lines of a normal file.
- An object can be loaded when a request arrives for it; or all of the required objects can be loaded when the first request is made. For example, in the bank application, an account object can be loaded when an invocation is made on it, or all of the accounts controlled by a bank can be loaded when the bank, or any of its accounts, is first interacted with.

- An object can be saved to the persistent store at the termination of the process, or it can be saved before that time: for example, at the end of the method call that caused it to be loaded, or if the object has not been used for some period of time.

Many different arrangements are possible for the loaders themselves, for example:

- A process can have a single loader to handle all of the interfaces that it supports. However, it is difficult to maintain such a loader for many interfaces.
- A process can have one loader to handle each interface, or each separate hierarchy of interfaces.

If one loader per interface is used, each loader's `load()` method is called in turn until one indicates that it can load the target object. Although this approach is simple to implement, such a linear search may be inefficient if a process handles a large number of interfaces. One efficient mechanism is to install a master loader, with which the other loaders can register. Each registration gives some key indicating when the registering loader's `load()` method is to be called by the master loader; a key can be a marker prefix and an interface name.

Another reason for having more than one loader is that a process may use objects from separate subsystems—each of which installs its own loader(s). These loaders must be able to distinguish requests to load their own objects. You can avoid confusion if the subsystems handle disjoint interfaces, since the interface name is passed to a loader; however, some co-operation between the subsystems is required if they handle the same interfaces, or interfaces which have a common base interface. Each subsystem must be able to distinguish its objects based on their markers or their persistent state.

If `I1` is a base interface of `I2` and `I3`, the objects of interfaces `I2` and `I3` must be distinguishable to avoid confusion when "`I1`" is passed as an interface name to `load()`.

In particular, the subsystems must choose disjoint markers.

Disabling the Loaders

On occasion, it is useful to be able to disable the loaders for a period. If, when binding to an object, the caller knows that the object already loaded *if* it exists, it might be worthwhile to avoid involving the loaders if the object cannot be found.

You can disable the loaders by calling the following method:

```
// Java
// In package IE.Iona.OrbixWeb.CORBA
// in class BOA.
public boolean enableLoaders (boolean b)
```

on the `_CORBA.Orbix` object, with a `false` parameter value. This returns the previous setting; the default is to have loaders enabled.

25

Locating Servers at Runtime

When `bind()` *is called with a null host name, OrbixWeb uses the locator to find the target object in the distributed system. This chapter describes the default locator supplied with OrbixWeb and explains how to replace it with a user-defined locator implementation.*

The Default Locator

The default OrbixWeb locator mechanism searches for a server using the following sequence of steps:

1. The locator first attempts to contact an OrbixWeb daemon process at the local (client) host. If no such process exists, the location attempt fails.
2. The locator invokes the method `lookUp()`. This contacts the local OrbixWeb daemon and requests a list of host names for the specified server name.

   This list is generated from the `Orbix.hosts` and `Orbix.hostgroups` files. If there is no entry for the requested server in these files, the OrbixWeb daemon connects to a daemon specified in an `IT_daemon` entry in the `Orbix.hosts` file, and calls `lookUp()` on this daemon (less one hop). This daemon then in turn returns a sequence of hosts that the server may be registered on from its `Orbix.hosts` and `Orbix.hostgroups` files.

3. The host names returned by the daemon are arranged in a random order. OrbixWeb then iterates through this list, attempting to verify the registration of the server at each host in turn. The locator returns the first host at which the specified server is registered.

If the client is an applet and `org.omg.CORBA.ORB.init(java.applet.Applet app, java.util.Properties props)` has been invoked, step 1 above uses the applet's codebase host instead of the local host.

If the location attempt fails, the `bind()` call also fails and throws an OrbixWeb system exception. The location attempt succeeds when it locates a host at which the server name passed to `bind()` has been registered. However, this does not guarantee that the `bind()` call itself will succeed. The `bind()` method requires additional criteria, such as successful launching of the server and location of the specified object within the server.

For successful operation of the default locator, you should specify server names and corresponding target hosts in advance. You must configure the default locator with the local OrbixWeb daemon process, which manipulates the locator configuration files. Refer to the chapter "OrbixWeb Configuration" in the *OrbixWeb Programmer's Reference* for more details.

The lookUp() method

The `lookUp()` method is a crucial part of the implementation of the OrbixWeb default location mechanism. This method is responsible for nominating a list of candidate host names at which the server should be sought. The signature of `lookUp()` (as defined in class `IE.Iona.OrbixWeb.Features.locatorClass`) is as follows:

```
// Java
public String[] lookUp(String ServiceName,
        int MaxHops, Context ctx);
```

Refer to "Parameters to lookUp()" on page 478 for more details on this method.

Unlike the location mechanism in some versions of Orbix, the OrbixWeb default locator does not use a publicly accessible default locator object to call the method `lookUp()`. Consequently, an OrbixWeb client cannot call the default locator `lookUp()` method directly. However, you can implement this functionality and this is discussed in the next subsection.

The object `_CORBA.locator` (defined in package `IE.Iona.OrbixWeb`) is only used to allow you to override the default `lookUp()` implementation, and is assigned `null` unless explicitly replaced, as described in "Writing a New Locator" on page 477.

Default lookUp() functionality

Although you do not normally have to make explicit use of the `lookUp()` method (since it is used implicitly through calls to `bind()`), it may sometimes be useful to do so. A direct call to the `lookUp()` method invoked by the OrbixWeb default locator is not possible, but you can easily copy this functionality.

The default `lookUp()` implementation uses the `IT_daemon` IDL interface, which is implemented by the OrbixWeb daemon. To copy the behaviour of `lookUp()`, you need to bind to the local OrbixWeb daemon process and invoke the `IT_daemon::lookUp()` operation. The IDL signature of this operation is:

```
// IDL
boolean lookUp (in string service,
        out stringSeq hostList,
        in octet hops, in string tag);
```

You can code the operation invocation as follows:

```
// Java
// in class Client

import IE.Iona.OrbixWeb.Activator.IT_daemon;
import IE.Iona.OrbixWeb.Activator.
        IT_daemonPackage.stringSeqHolder;
import IE.Iona.OrbixWeb.Activator.IT_daemonHelper;
import org.omg.CORBA.SystemException;
...

IT_daemon dRef;
String service;
String tag = null;
byte hops;
stringSeqHolder hostList = new stringSeqHolder();

// initialize server name
service = "MyServer";

// initialize number of locator hops
hops = 5;

try {
   // bind to Orbix daemon
   dRef = IT_daemonHelper.bind ();
```

```
      // invoke lookUp() operation

      dRef.lookUp (service, hostList, hops, tag);
   }
   catch (SystemException se) {
      ...
   }

   if (hostList.value.length > 0) {
      int i;

      for (i=0; i<hostList.value.length; i++)
         System.out.println (hostList.value[i]);
   }
   else {
      // server not found in configuration file ...
   }
```

Each string in the sequence of strings returned by operation `lookUp()` gives the name of a host on which the specified server may be registered. The `lookUp()` operation returns an empty sequence if no host names can be found for the specified server. If the call succeeds, the program can choose any of the returned host names, or perhaps iterate over the host names, attempting to bind to the required object at each in turn. If an exception is raised on one of the binds, this indicates an error such as the host not being available.

The default implementation of the locator randomizes the sequence before returning it. This is a basic technique in load balancing to avoid swamping any one server.

The `hops` parameter to `lookUp()` specifies the maximum number of hops that can be used to fulfil a request, thereby limiting the number of hosts involved in a search. The `bind()` method uses the value `IT_LOCATOR_HOPS` configuration value. You can change this value if you wish to modify how `bind()` uses `lookUp()`. Explicit calls to `lookUp()` can specify any `byte` value. The constant value `_CORBA._MAX_LOCATOR_HOPS` is used if a greater value is specified.

You can set the hops configuration variable as follows:

```
ORB.setConfigItem ("IT_LOCATOR_HOPS", "5");
```

The `tag` parameter is simply a string used in daemon diagnostic messages and can generally be assigned the value `null`.

Writing a New Locator

If the search facility provided by the OrbixWeb default locator is not appropriate, or if it needs to be augmented for a given application, you can install an user-defined alternative locator by:

1. Defining a derived class of `locatorClass`.
2. Creating a single instance of the new class.
3. Assigning the pointer `_CORBA.locator` to point to that instance.

The default value of `_CORBA.locator` is `null`. This indicates that the default locator algorithm should be used where appropriate.

The locator `lookUp()` method is passed the name of the server being sought. It should return a list of names of hosts on which that server is registered in the Implementation Repository. It is often advisable for a locator to randomize the sequence before returning it.

Class `locatorClass` is defined as follows:

```
// Java
package IE.Iona.OrbixWeb.Features;

public class locatorClass {
   public String[] lookUp(
      String ServiceName, int MaxHops,
      Context ctx);
}
```

The parameters to `lookUp()` are described as follows:

Parameters to lookUp()

`ServiceName`	The name of the server being sought.
`MaxHops`	This is interpreted as the maximum number of machines to search for the required server. You should retain an interpretation similar to this one in a user defined locator if you want to use it without changing client code that explicitly calls `lookUp()`.
`context`	A context parameter. This allows a client to pass extra information to the locator: for example, constraints on how to search for the server. A *trader* is an example of where this is important. You can use the context parameter to define properties to be used when deciding between a set of servers with the same name.

26

Opaque Types

OrbixWeb provides an extension to IDL that allows you to define opaque data types. Opaque data types can be passed by value through an IDL definition. This chapter describes how to use opaque data types with OrbixWeb.

In accordance with the CORBA standard, OrbixWeb objects are passed to and from IDL operations *by reference*. OrbixWeb objects are described by an interface which is defined in IDL. These objects are created in a server. Object references rather than actual copies of the objects are passed to clients.

This model applies to the majority of applications that use an ORB. However, in some cases, you may wish to pass objects across a CORBA IDL interface *by value* rather than by reference. Passing an object by value means that the internal state of the object is included in an operation parameter or return value. A copy of the object is constructed in the process.

In addition, there has been demand for a mechanism that allows existing objects to be passed across an IDL interface without having to retrospectively define IDL interfaces for these objects. Such a mechanism allows the integration of IDL types with non-IDL data types within a CORBA environment.

Opaque types address both of these issues. A new opaque keyword identifies a IDL data type as *opaque*. This means that nothing is known at the IDL level. A type defined to be opaque behaves like an interface type. This means that it may be passed as a parameter or return value to an IDL operation. It may also be used as an attribute type or as a member of a struct or exception.

An opaque type is always passed to and from IDL operations by value. You must supply the following:

- A Java class which implements the opaque object.
- The opaque's Helper class which implements the stream based marshalling and unmarshalling of the opaque object.

Possible Alternative Solutions

As outlined in the previous section, IONA's approach to passing objects between client and server processes by value is to introduce a new type constructor at the IDL level.

It is possible to achieve similar results without extending the IDL language. One solution to transmitting an object by value is to define its state in an IDL `struct` definition. This solution is unsatisfactory for two reasons: first, you are forced to separate state information from interface information; second, you must make explicit in the IDL definition information that properly belongs to the implementation.

A second solution is to pass an object's state information in binary form, as a `sequence<octet>`. This mechanism does not make explicit the type of the information transmitted, so it does not violate the privacy of the object. However, no marshalling or unmarshalling is performed on a `sequence<octet>`, so byte-swapping and other data-conversion becomes the responsibility of the programmer. Further, in stripping the interface of type information, the ORB assumes the role of an RPC package.

Using Opaque Types

This section demonstrates how to use the opaque mechanism to pass a user-defined type by value in IDL operations. The sample code described in this section is available in the `demos/Date` directory of your OrbixWeb installation.

IDL Definition

The example used here defines an IDL interface `Calendar` that makes use of the opaque type `Date`. The IDL definitions are as follows:

```
// IDL
// In file calendar.idl.

opaque Date;

interface Calendar {
   // Today's date.
   readonly attribute Date today;

   // Length of time from given date until today.
   unsigned long daysSince(in Date d);
};
```

The opaque data type is introduced by the keyword `opaque`, denoting a new IDL type. An opaque type may be defined at file level scope or within a module, at the same level as an interface definition. In this example, the new `Date` type is used as an attribute type and as an `in` parameter.

Compiling the IDL Definition

You can compile IDL definitions using the -K switch, as follows:

```
idl -jPopaqueDateDemo -K calendar.idl
```

`opaque` is not a keyword in CORBA IDL. The `-K` switch to the IDL compiler indicates that support for opaque types is required.

Mapping of Opaque Types to Java

The following template classes are generated by the IDL compiler:

```
// the date class
_DateTemplate.java

// the Holder class
_DateHolderTemplate.java

// the Helper class
_DateHelperTemplate.java
```

Implementing the Opaque Type

The generated file `_DateTemplate.java` contains the template `Date` implementation class. You should change the name of `_DateTemplate.java` to `Date.java`. The following is an example implementation for the `Date` class:

```
// Java
// In file _DateTemplate.java.

package opaqueDateDemo;

public class Date  {

   public Date () {}

   public Date (int day, String month, int year) {
      this.day = day;
      this.month = month;
      this.year = year;
   }
   public String toString() {
      return("Date ==> " + day + " " + month + " " + year);
   }

   public int day;
   public String month;
   public int year;
}
```

The Helper Class

The generated file _DateHelperTemplate.java contains the code you must use to stream information into and out of the Date objects.

This involves implementing read() and write() methods to marshal and unmarshal the objects. The org.omg.CORBA.portable.InputStream and OutputStream interfaces are use for this:

```
// Java
// In file _DateHelperTemplate.java.

package opaqueDateDemo;

import IE.Iona.OrbixWeb._OrbixWeb;

public class DateHelper {

   public static Date read
      (org.omg.CORBA.portable.InputStream _stream) {
         Date value = new Date();
         value.day = _stream.read_long();
         value.month = _stream.read_string();
         value.year = _stream.read_short();
         return value;
   }

   public static void write
      (org.omg.CORBA.portable.OutputStream _stream, Date value) {
         _stream.write_long(value.day);
         _stream.write_string(value.month);
         _stream.write_long(value.year);
   }
   ...
}
```

You should change the name of_DateHelperTemplate.java to DateHelper.java.

The Holder Class

The generated file _DateHolderTemplate.java is the Holder for Date. You can avoid implementing the marshalling again by invoking the Helper class read() and write() methods as follows:

```
// Java
// In file _DateHolderTemplate.java.

package opaqueDateDemo;

import IE.Iona.OrbixWeb._OrbixWeb;

public final class DateHolder
   implements org.omg.CORBA.portable.Streamable {

   public Date value;

   public DateHolder() {
      value = new Date();
   }

   public DateHolder(Date value) {
      this.value = value;
   }

   public void _read
      (org.omg.CORBA.portable.InputStream _stream) {
      DateHelper.read(_stream);
   }

   public void _write
      (org.omg.CORBA.portable.OutputStream _stream) {
      DateHelper.write(_stream, value);
   }
   ...
}
```

You should also change the name of this file from _DateHolderTemplate.java to DateHolder.java.

Refer to the demos/Date directory of your OrbixWeb installation for an example client/server application that uses the Date type.

27

Transforming Requests

This chapter describes how you can modify the data buffers containing OrbixWeb operation call information immediately before and after transmission across the network.

In OrbixWeb, an operation invocation or an operation reply is transmitted between a client and a server in a `org.omg.CORBA.Request` object. Using the Dynamic Invocation Interface, an `org.omg.CORBA.Request` is explicitly created. A static invocation results in the implicit creation of a `org.omg.CORBA.Request` object.

This chapter describes how you can modify an OrbixWeb `Request` data buffer and allow a client or server process to specify what modifications to the buffer should occur when requests or replies are transmitted to other processes. The ability to modify this data just before its transmission, or just after its reception means that you can add additional information to the data stream. For example, you can add information identifying the participants in the communication or encrypt the data stream for security purposes. The process of modifying the data buffer is known as *transforming* the data buffer.

The functionality provided by transformers is at a lower level than that provided by filters, since it allows access to the actual data buffer transmitted in a `Request`.

Transforming Request Data

You can transform a `Request` data buffer using a *transformer object*. To obtain a new transformer object, perform the following steps:

1. Define a class which inherits from the class `IE.Iona.OrbixWeb.Features.IT_reqTransformer`.
2. Create an instance of this class.
3. Register this instance with the OrbixWeb runtime.

   You can register the transformer object so that it performs transformations on all communications to and from the process that contains the transformer object. Alternatively, you can register it so that transformations are performed only on communications to and from a particular server on a particular host that contains the transformer.

Note: Because transformations are applied when an operation invocation leaves or arrives at an address space, no transformations are applied when the caller and invoked object are collocated.

The IE.Iona.OrbixWeb.Features.IT_reqTransformer Class

The `IT_reqTransformer` class defines the interface to transformer objects. This class is defined as follows:

```
// Java

package IE.Iona.OrbixWeb.Features;

public class IT_reqTransformer {

   public boolean transform(octetSeqHolder data,
           String host,
           boolean is_send,
           org.omg.CORBA.Request req) {
      return true;
   }
```

```
        public String transform_error() {
            return null;
        }
    }
```

A class derived from `IT_reqTransformer` can access a data buffer just before transmission and can therefore manipulate or transform the data as required. The derived class must, at least, override the `transform()` method. Refer to the *OrbixWeb Programmer's Reference* for full details of the `IT_reqTransformer` class.

The `transform()` method is called by OrbixWeb immediately prior to transmitting the data in a `Request` out of an address space and immediately subsequent to receiving a `Request` from another address space. The derived class can allocate new storage to handle any alteration in the data size caused by the transformation.

The `transform()` method can indicate that a `org.omg.CORBA.COMM_FAILURE` system exception should be raised by OrbixWeb by returning `false`.

A derived class may implement the `transform_error()` method to return a string containing suitable error text.

The `req` parameter in the `transform()` method holds a reference to the `Request` object when an outgoing `transform()` is called. This has a value of null for all incoming transform operations.

Registering a Transformer

OrbixWeb provides two methods to register a transformer object (an instance of `IT_reqTransformer`). You can call both on the `ORB` object:

- `setMyReqTransformer()`
- `setReqTransformer()`

setMyReqTransformer()

This method is defined as follows:

```
// Java
// In class IE.Iona.OrbixWeb.CORBA.ORB

IT_reqTransformer setMyReqTransformer(
   IT_reqTransformer transformer)
```

`setMyReqTransformer()` registers a transformer object as the default transformer for all `Requests` entering and leaving an address space.

setReqTransformer()

This method is defined as follows:

```
// Java
// In class IE.Iona.OrbixWeb.ORB.

void setReqTransformer(
        IT_reqTransformer transformer,
        String server,
        String host)
```

`setReqTransformer()` registers a transformer object for all `Requests` destined for a specific server and host and for all `Requests` received from a specific server and host. You can call this method more than once to register different server/host pairs.

A transformer registered using `setReqTransformer()` overrides any default transformer registered with `setMyReqTransformer()`.

Note: At most, one transformation is applied to any `Request`—the default transformation registered with `setMyReqTransformer()` or overriding specific transformation registered with `setReqTransformer()`.

An Example Transformer

This section presents a simple example of a transformer that adds the name of the sending host to a `Request`'s buffer when sending a `Request` out of a process and removes the host name from a `Request`'s buffer when receiving a `Request` containing an operation reply.

The transformer is implemented as follows:

```
// Java
...

public boolean transform(octetSeqHolder data,
             String host,
             boolean is_send
             org.omg.CORBA.Request req)
{
   if (is_send) {
      byte[] buf = new byte[data.value.length +
                host.length() + 4];

      // insert the host name length
      buf[0] = (byte)((host.length() >> 24) &
                0x000000ff);
      buf[1] = (byte)((host.length() >> 16) &
                0x000000ff);
      buf[2] = (byte)((host.length() >> 8) &
                0x000000ff);
      buf[3] = (byte)(host.length() & 0x000000ff);

      // insert the host name
      System.arraycopy(host.getBytes(), 0, buf,
                4, host.getBytes().length);

      // add the OrbixWeb data buffer
      System.arraycopy(data.value, 0, buf, 4 +
                host.length(), data.value.length);
      data.value = buf;
   }
   else {
      // extract the host name length
      int l = ((((int)data.value[0]) << 24) &
             0xff000000) |
```

```
                ((((int)data.value[1]) << 16) &
                   0x00ff0000) |
                  ((((int)data.value[2]) <<  8) &
                   0x0000ff00) |
                   (((int)data.value[3]) & 0x000000ff);

            // extract the host name
            String h = new String(data.value, 4, l);
            int len = data.value.length - h.length() - 4;

            // extract the OrbixWeb data buffer
            byte[] buf = new byte[len];
            System.arraycopy(data.value, 4 +
               host.length(), buf, 0, len);
            data.value = buf;
          }
         return true;
      }

      java.lang.String transform_error() {
         return "Error in Transformer";
      }

      // Create a Transformer:
      Transformer transformer = new Transformer();
```

The `transform()` method uses the parameter `is_send`. This indicates whether the `Request` is incoming or outgoing, to determine whether to add or remove the host name from the `Request`'s buffer.

Registering the Transformer

The following call registers this transformer as the default transformer for a client or server process:

```
ORB.setMyReqTransformer(transformer);
```

To register a transformer that acts on Requests going to or received from a specific server on a specific host, make the following call:

```
// Register a transformer that transforms data
// sent to or received from myServer on host
// alpha.

ORB.setReqTransformer(
     transformer,"myServer", "alpha");
```

28

Service Contexts

Service contexts provide a means of passing service-specific information as part of IIOP message headers. This chapter describes OrbixWeb APIs that allow you to register handlers that intercept IIOP requests and replies, and to store and retrieve service contexts.

A service context consists of a unique ID and a sequence of octets. Its structure can be outlined as follows:

```
// IDL
module IIOP {
   typedef unsigned long ServiceId;

   struct ServiceContext {
           ServiceId context_id;
           sequence<octet> context_data;
   };
   typedef sequence<ServiceContext> ServiceContextList;

};
```

The `context_id` is a unique ID by which a particular service context is recognized. The `context_data` octet sequence is the part of the context containing the data.

Note: Service contexts in OrbixWeb can only be used over IIOP.

The OrbixWeb Service Context API

The OrbixWeb API for service contexts comprises the following external interfaces:

- The `ServiceContextHandler` class.
- The ORB interfaces.
- The `ServiceContextList`.

ServiceContextHandler Class

The `ServiceContextHandler` class is the base class from which you derive handlers for a particular `ServiceContext`. Each handler has a unique ID. This corresponds to the ID of the particular `ServiceContext` used. You should register a handler on both the client and the server for each `ServiceContext`. Refer to "ORB Interfaces" on page 495 for more details.

The `ServiceContextHandler` base class has the following structure:

```
// Java

import IE.Iona.OrbixWeb.CORBA.Request;

class ServiceContextHandler {

   // Constructor registers the context ID
      public myServiceContext(int Context_Id) {
         super(Context_Id)
      }
      public boolean incomingRequestHandler(Request req);
      public boolean outboundRequestHandler(Request req);
      public boolean incomingReplyHandler(Request req);
      public boolean outboundReplyHandler(Request req);
};
```

ORB Interfaces

ORB APIs are provided to allow you to register the handler with the ORB. These APIs are defined as follows:

```
// Java

public class ORB {
   ...
   public void registerPerRequestServiceContext
                      (ServiceContextHandler CtxHandler);

   public void unregisterPerRequestServiceContext
                                      (int CtxHandler_Id);

   public void registerPerObjectServiceContext
                      (ServiceContextHandler CtxHandler,

                          org.omg.CORBA.Object HandledObject);
   public void unregisterPerObjectServiceContext

                      (int CtxHandler_Id,
                          org.omg.CORBA.Object HandledObject);
}
```

Per-Request Handlers

Registering a handler as per-request adds its request/reply handler methods to a `ServiceContextList` (SCL). The handler is then called at the appropriate point for the request.

Per-Object Handlers

Registering a handler as per-object also adds its request/reply handler methods to a `ServiceContextList`. The handler is then called for requests /replies associated with the specified target object.

ServiceContextList

A `ServiceContextList` is a field in a IIOP message header containing all the service context data associated with a request or reply.
A `ServiceContextList` is implemented as a sequence of `ServiceContexts`. `ServiceContextLists` support both per-object and per-request service context handlers.

Using Service Contexts in OrbixWeb Applications

Service contexts in OrbixWeb are based on two models:

Service Context per-request	In this model service contexts are handled on all requests and replies entering and leaving an ORB.
Service Context per-object	In this model only service context information is handled for requests and replies going to or coming from a particular object.

ServiceContext Per Request Model

This section gives an overview of implementing per request service contexts in OrbixWeb applications.

Client-Side

To add service context information to all requests leaving a client application, do the following:

1. Call the `enableServiceContext()` method on the ORB to enable `ServiceContexts`.
2. In the user code, derive a class from the base class `ServiceContextHandler`. For example, `myServiceContextHandler`.
3. Create an instance of this class within the client, and pass it a unique `ServiceContext_Id`.

4. Register this handler instance with the ORB using the following method:

```
void registerPerRequestServiceContextHandler
                (ServiceContextHandler myHandler)
```

   This registration means, for example, if any outgoing requests leave the client, the following method is called:

```
myServiceContextHandler.outboundRequestHandler
                                (Request req)
```

   This method takes the request that caused the invocation as a parameter. The request is interrogated by the user handler class showing the operation name.

   Similarly, for incoming requests `incomingReplyHandler()` is called.

5. Create a new instance of `ServiceContext` in the user code of the handler.
6. Populate the `context_data` part of the `ServiceContext` with information, and add it to the `ServiceContextList`.

   This `ServiceContextList` is marshalled with the request message and is passed across the wire to the server.

Server-Side

The server side design is similar to the client side. It creates and registers handlers, and re-implements the methods from the `serviceContextHandler` class. To receive service context information from all requests entering a server, do the following:

1. Call the `enableServiceContext()` method to on the ORB enable `ServiceContexts`.
2. In the user code, derive a class from the base class `ServiceContextHandler`. For example, `myServiceContextHandler`.
3. Create an instance of this class within the server passing it the `ServiceContext_id`. You can use the same code on both the server and client sides.
4. Register this handler instance with the ORB using

```
void registerPerRequestServiceContextHandler
                (ServiceContextHandler myHandler)
```

   This registration means that when a request comes into the server address space, the `ServiceContextList` in the request header is

unmarshalled. This means that only the relevant handlers are called via the following method:

```
public boolean incomingRequestHandler(Request req);
```

If there is a `ServiceContext` in the request header list that has the same ID as the registered handler, the `incomingRequestHandler()` method is called.

5. Using the `incomingRequestHandler()` method, take a copy of the `ServiceContext` required, and extract the needed information, calling the necessary code. This information can then be processed.

After the handler has returned, and all other `ServiceContext` handlers have completed, the request continues as normal.

Note: Replies are treated the same as requests. They activate the `outboundReply()` and `incomingReply()` handlers in the same way.

Example ServiceContextHandler

Given the following IDL definition:

```
//IDL
struct myStruct {
   long current;
   string message;
};
```

you can write a `ServiceContextHandler` to send and receive `myStruct` objects across the wire, as follows:

```
//java

import IE.Iona.OrbixWeb.Features.ServiceContextHandler;
import IE.Iona.OrbixWeb.CORBA.Request;

public class myServiceContextHandler
   extends ServiceContextHandler {
   long num = 0;
   public myServiceContextHandler(int id) {
      super(id);
   }
```

```
    public  boolean outboundRequestHandler(Request req) {
       System.out.println
          ("Attempting to add Service Context list to outgoing
             Request \n" + "\ttarget \t" + req._target() +
             "\tcalling  \t" + req.operation() );

       myStruct s = new myStruct (++num, "this is message number"
                                                          + num);
       Any a = new Any(_CORBA.IT_INTEROPERABLE_OR_KIND);
       myStructHelper.insert(a,s);

       ServiceContext sc = new Servicecontext();
       sc.context_id  = _getID();
       sc.context_data = a.value();
       req.addServiceContext(sc);
       return true;
    }

    public boolean incomingRequestHandler(Request req) {
       System.out.println
          ("attempting to extract data from Service Context
             List on incoming Request \n" + "\ttarget \t" +
             req._target() + "\tcalling  \t" + req.operation());

       Servicecontext sc = req.
       getServiceContext( _getID());

       Any a = new Any(myStructHelper.type(), sc.context_data,
                                  sc.context_data.length, true);
       myStruct s = myStructHelper.extract(a);

       System.out.println
          ("Extracted the following data from Service Context
             List on incoming Request \n" + "\tID \t\t" +
             sc.context_id + "\tstruct num \t" + s.num +
             "\tstruct msg \t" + s.message  );
       return true;
    }
}
```

ServiceContext Per-Object Model

This section gives an overview of implementing per object service contexts in OrbixWeb applications.

Client-Side

To add `ServiceContexts` to requests leaving the client for a particular object you must also create and register handlers. This involves the following:

- The `registerPerObjectServiceContextHandler()` method returns the handler and object reference.
- The object reference is stored in a `Vector` array.
- Each `ServiceContext` in the `ServiceContextList` has the same ID as one of the handlers registered for that object.
- Only one `ServiceContextList` is marshalled and sent across on the wire.

Server-Side

To receive `ServiceContexts` from requests entering the server for a particular object you must create and register handlers. The following stages are involved:

- An object reference is obtained and stored in a `Vector` array.
- The `incomingRequest()` method is called for any `ServiceContext` IDs that correspond to any of the handlers registered.

Main Components

The `ServiceContext` per-request and `ServiceContext` per-object models comprise a number of common components. This section defines each component and explains how these components interact.

ServiceContextHandler

This base class allows users to define their own handlers for a particular `Context_Id`. For each `ServiceContext` you wish to handle, there is a handler registered on both the client and on the server. Each handler is recognized by its ID which corresponds to the ID of the `ServiceContext` it handles.

The ServiceContextHandler base class includes the following methods:

- **incomingRequestHandler()**

  This method is called when an incoming request arrives in a server at the point where the ServiceContextList has been unmarshalled. It accesses the unmarshalled ServiceContextList, passing the appropriate Context_Id required to access a specific ServiceContext.

- **outboundRequestHandler()**

  This method is called when an outgoing request is being marshalled in the client. It can add a ServiceContext to the ServiceContextList for marshalling.

- **incomingReplyHandler()**

  This method is called when an incoming reply arrives in a client at the point where the ServiceContextList has been unmarshalled. It accesses the unmarshalled ServiceContextList, passing the appropriate ServiceContext_Id required to access a specific ServiceContext.

- **outboundReplyHandler()**

  This method is called when an outgoing reply is being marshalled in the server. It can add a ServiceContext to the ServiceContextList for marshalling.

PerRequestServiceContextHandler

This is a SerivceContextHandler that has been registered as a handler for all requests on the client or server side. The user derives from the base class, and registers the handler. The handler is recognised by its ID. This corresponds to the ID of the ServiceContext it handles.

PerObjectServiceContextHandler

This is a ServiceContextHandler that has been registered as a handler for all requests to a particular object on the client or server side. The user derives from the base class and registers the handler. The handler is recognised by its ID which corresponds to the ID of the ServiceContext it handles.

PerRequestServiceContextHandlerList

This is a list of service context handlers. For all requests or replies leaving an address space, all outbound methods in all handlers are called. This is because you do not know which `ServiceContext` to add to each request.

For all incoming requests or replies in the client address space, only the incoming methods of the handlers with IDs corresponding to actual `ServiceContexts` are called.

Similarly, on the server-side, for all outgoing requests or replies, only the outgoing methods of the handlers whose IDs corresponds to actual `ServiceContexts` in the request or reply header are called.

PerObjectServiceContextHandlerList

This works the same way as `PerRequestServiceContextHandlerList` except that only requests and replies relating to a particular object are both tagged and have their `ServiceContext` data investigated. `PerRequestServiceContextHandlerList` is actually a list indexed by both the context ID and the `omg.org.CORBA.Object` it references.

Service Context Handlers and Filter points

Service context handlers also interact with OrbixWeb filter points. In OrbixWeb, there are ten filter points, including the in reply and out reply failure filter points. Refer to Chapter 22, "Filters" for more details. The service context mechanism provides four more points for interaction with requests and replies in a typical invocation.

Figure 37 shows the position of the `ServiceContextHandlers` in an invocation, in the subsequent reply, and also the order in which they are called.

If an exception is thrown in any of the `outRequest()` pre or post marshall filter points on the client side, the `incomingReplyHandler()` is not called.

Oneway calls do not return anything, thus they do not call the client side `inboundReplyHandler()`.

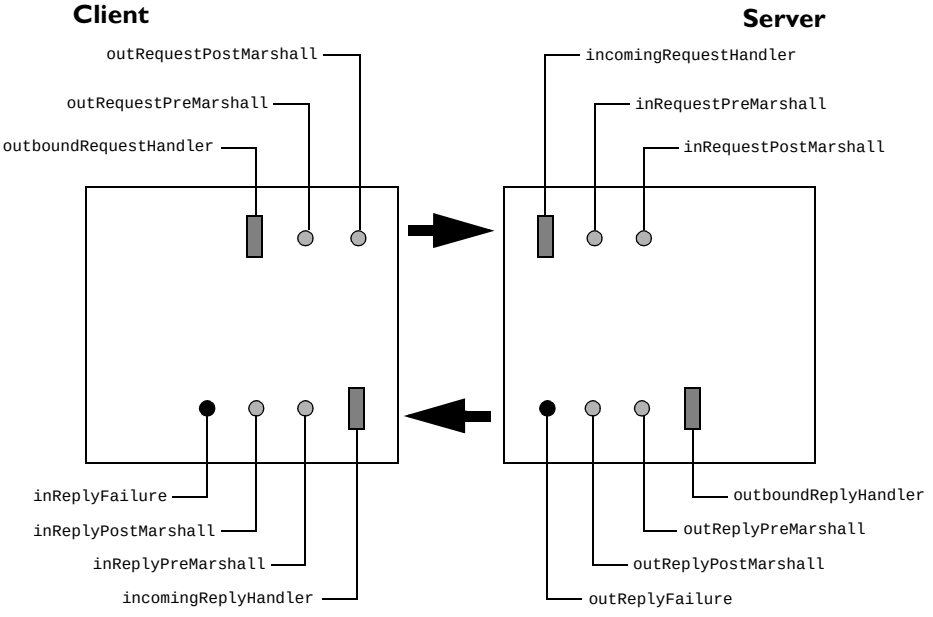

Figure 37: *`ServiceContext` Handlers and Filter Points*

Appendix A
IDL Compiler Switches

This appendix describes the command-line switches to the IDL Compiler.

The IDL Compiler supports the following switches:

-C	Specify to the OrbixWeb IDL pre-processor that it should not filter out comments. Comments are filtered out by default. This switch is often used with `-E`.
-D <name>	Pre-define the macro `name` to be `1` within the IDL file.
-D <name>=<definition>	Pre-define the macro `name` to be `definition`.
-E	Only run the OrbixWeb IDL pre-processor. Do not pass the output of the pre-processor to the OrbixWeb IDL compiler, but output the pre-processed file to standard output. By default, the output of the OrbixWeb IDL pre-processor is sent to the OrbixWeb IDL compiler.
-F	Generate per-object filtering code.
-flags	Display the command-line usage summary.
-I <directory>	Specify an include file directory for use with IDL include directives of the form `#include<filename>`. You can specify more than one `-I` switch.

-jc	Generate support for client-side functionality only. By default, the IDL compiler generates both client-side and server-side support. This involves the creation of several server-specific source files that are not required by client programmers. This switch suppresses the generation of these files.
-jNoC	Specify that the generated constructors for TIE and Implbase classes do not implicitly call `_CORBA.Orbix.connect()`. The default is that the generated constructors implicitly call `_CORBA.Orbix.connect()`. If this switch is used an application must explicitly connect the newly-created implementation object before use.
-jO <directory>	Specify a target directory for the file structure output by the IDL compiler. The directory path may be absolute or relative. The default directory for IDL compiler output is `java_output`.
-jOMG	Ensure the generated code is OMG-mapping compliant by suppressing the addition of OrbixWeb-specific functionality. This functionality includes `bind()` and additional constructors that require `marker`, `loader` or `orb` parameters. Calling this switch also has the same effect as calling `-jNoC`.
-jP [<package> | <module>=<package>]	Specify a Java package name within which all IDL generated Java code is placed, or an IDL module which should be mapped to a specific package name. By default, generated code is placed within the global package, so the use of this switch is generally recommended to avoid naming clashes.
-jQ	Generate support for the `equals()` method in all IDL produced Java classes.

-K	Required if the IDL file uses the opaque type specifier.
-m <IIOPonly>	Generate marshalling code for the CORBA Internet Inter-ORB Protocol (IIOP) only. By default, code generated by the IDL Compiler supports both IIOP and the Orbix protocol.
-N	Specify that the IDL compiler is to compile and produce code for included files (files included using the `#include` directive). Without the `-N` switch, included files are compiled but no code is output. The use of the `-N` flag is not encouraged as it complicates the use of the Interface Repository. The `-N` flag also has the restriction that the compilation must be invoked from the same directory as the root IDL file to retain compatibility with the Interface Repository server.
-U <name>	Do not pre-define the macro `name`. If `-U` is specified for a macro name, that macro name is not defined even if `-D` is used to define it.
-v	Print version information. The version information includes the IDL compiler release and the target JDK version number.

Note: It is necessary to process each IDL file through the IDL compiler. Inclusion of an IDL file in another (using `#include`) is not sufficient to produce output for the included file (unless the `-N` switch is specified to the compiler). Otherwise, Java code generation would occur more than once for a file that was included in more than one file.

Index

D

E

F

N

O

P

R

S

T

U

W