Orbix 3.3.14

Programmer’s Guide Java Edition

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK

http://www.microfocus.com

2017-04-25

Contents

Preface

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

Audience

The ***Orbix Programmer's Guide Java Edition*** and the ***Orbix Programmer's Reference Java Edition*** are intended for use by application programmers and designers wishing to familiarize themselves with CORBA distributed programming and its application in the Java environment. The ***Orbix Administrator's Guide Java Edition*** describes how to use various command line and GUI tools during Orbix Java operation. These guides assume that you are familiar with the Java programming language.

Organization of the Orbix Java Edition Documentation

The complete Orbix Java Edition documentation set includes the following manuals:

- The ***Orbix Programmer's Guide Java Edition*** provides a complete guide to Orbix Java programming.
- The ***Orbix Programmer's Reference Java Edition*** provides an exhaustive reference for the Orbix Java application programming interface (API).
- The ***Orbix Administrator's Guide Java Edition*** explains how to configure and manage the components of the Orbix Java environment using the command line and Orbix Java GUI tools.

Organization of this Guide

The ***Orbix Programmer's Guide Java Edition*** is divided into the following five parts:

Part I "Getting Started"

This part of the guide introduces basic CORBA concepts and introduces Orbix Java.

Part II "CORBA Programming with Orbix Java"

Part II provides a description of developing CORBA programs in Java using Orbix Java.

This part of the guide provides an outline of the CORBA Interface Definition Language (IDL) and the standard Object Management Group (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 Orbix Java Programs"

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

Part IV "Advanced CORBA Programming"

This part of the guide explains more advanced features of Orbix Java 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 V "Advanced Orbix Java Programming"

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

Part VI "Appendix"

This contains an appendix listing the command-line options to the Orbix IDL compiler.

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.

Contacting Micro Focus

Our Web site gives up-to-date details of contact numbers and addresses.

Further Information and Product Support

Additional technical information or advice is available from several sources.

The product support pages contain a considerable amount of additional information, such as:

- The WebSync service, where you can download fixes and documentation updates.
- The Knowledge Base, a large collection of product tips and workarounds.
- Examples and Utilities, including demos and additional product documentation.

To connect, enter http://www.microfocus.com in your browser to go to the Micro Focus home page.

Note:
Some information may be available only to customers who have maintenance agreements.

If you obtained this product directly from Micro Focus, contact us as described on the Micro Focus Web site, http://www.microfocus.com. If you obtained the product from another source, such as an authorized distributor, contact them for help first. If they are unable to help, contact us.

Information We Need

However you contact us, please try to include the information below, if you have it. The more information you can give, the better Micro Focus SupportLine can help you. But if you don't know all the answers, or you think some are irrelevant to your problem, please give whatever information you have.

- The name and version number of all products that you think might be causing a problem.
- Your computer make and model.
- Your operating system version number and details of any networking software you are using.
- The amount of memory in your computer.
- The relevant page reference or section in the documentation.
- Your serial number. To find out these numbers, look in the subject line and body of your Electronic Product Delivery Notice email that you received from Micro Focus.

Contact information

Our Web site gives up-to-date details of contact numbers and addresses.

Additional technical information or advice is available from several sources.

The product support pages contain considerable additional information, including the WebSync service, where you can download fixes and documentation updates. To connect, enter http://www.microfocus.com in your browser to go to the Micro Focus home page.

If you are a Micro Focus SupportLine customer, please see your SupportLine Handbook for contact information. You can download it from our Web site or order it in printed form from your sales representative. Support from Micro Focus may be available only to customers who have maintenance agreements.

You may want to check these URLs in particular:

- http://www.microfocus.com/products/corba/orbix/orbix-3.aspx (trial software download and Micro Focus Community files)
- https://supportline.microfocus.com/productdoc.aspx. (documentation updates and PDFs)

To subscribe to Micro Focus electronic newsletters, use the online form at:

http://www.microfocus.com/Resources/Newsletters/infocus/newsletter-subscription.asp

Part I
Getting Started

In this part

This part contains the following:

Introduction to CORBA and Orbix Java

Orbix Java is a software environment that allows you to build and integrate distributed applications. Orbix Java 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 Orbix Java 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.

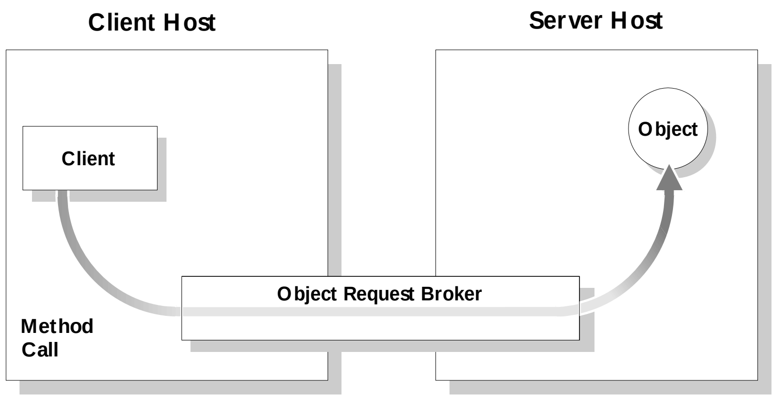


Figure 1: *The Object Request Broker*

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 code from IDL definitions. This Java code 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, 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.

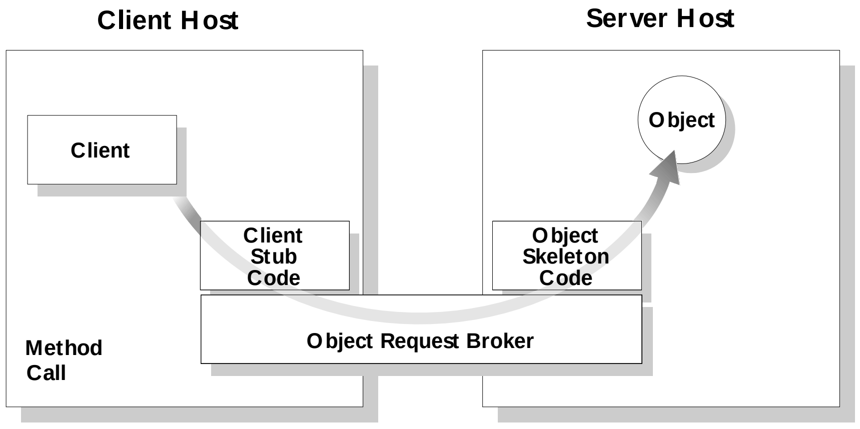


Figure 2: *Invoking on a CORBA Object*

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 6.

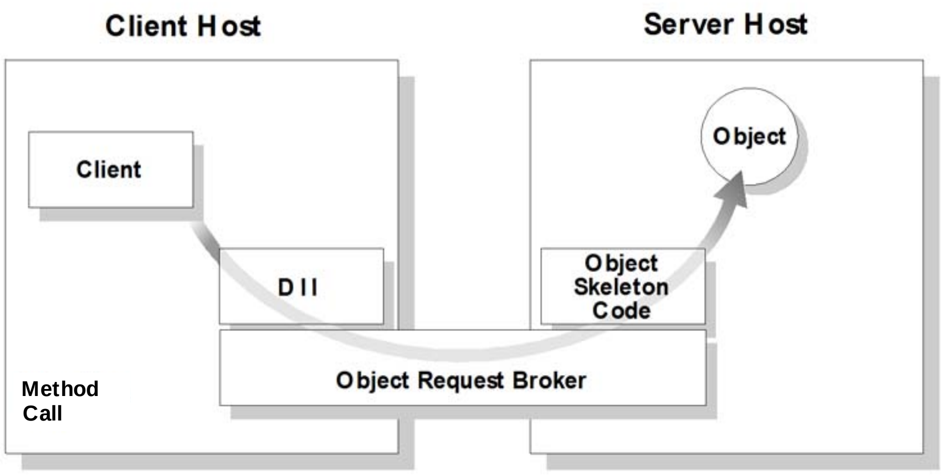


Figure 3: *Client Invoking a Method Using the DII*

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 6 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 Orbix Java servers.

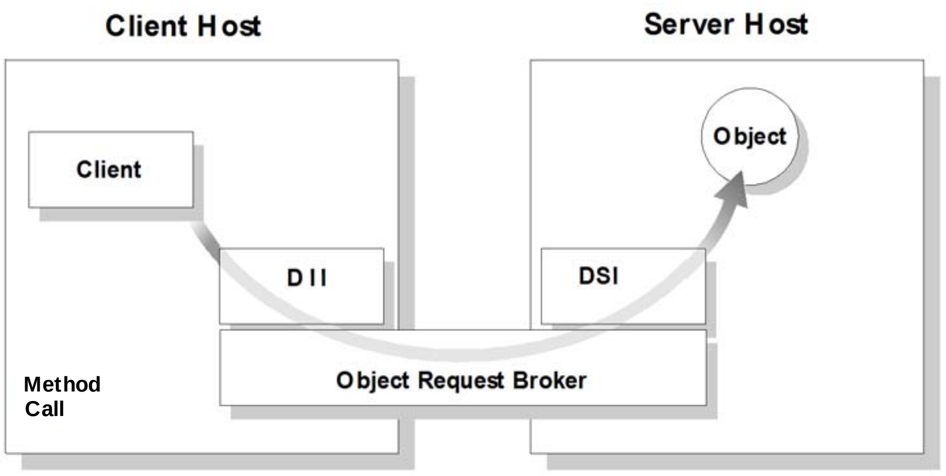


Figure 4: *Method Call Using the DII and DSI*

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.

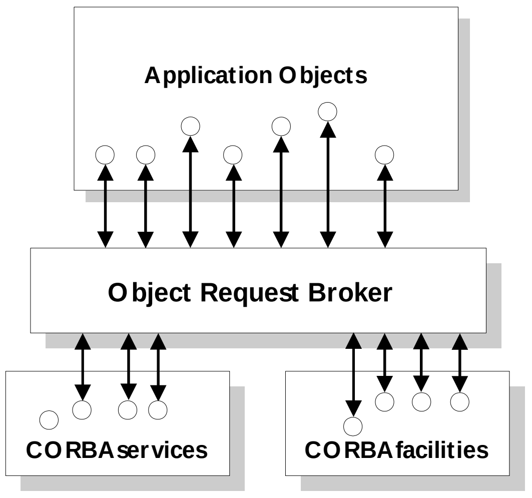


Figure 5: *The Object Management Architecture*

When writing Orbix Java applications, you might 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.

Orbix 3 implements several CORBAservices.

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 Orbix Java Implements CORBA

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

The components of Orbix Java are as follows:

- The *IDL compiler* parses IDL definitions and produces Java code that allows you to develop client and server programs.
- The *Orbix Java runtime* is called by every Orbix Java program and implements several components of the ORB, including the DII, the DSI, and the core ORB functionality.
- The *Orbix Java 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 *Interface Repository server* is a process that implements the Interface Repository.

Orbix Java also includes several programming features that extend the capabilities of the ORB. These features are described in Part IV "Advanced CORBA Programming".

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

Getting Started with Orbix Java

This chapter describes gives a brief overview of what is required to setup a Java development environment.

Prerequisites

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

- The Orbix developer's kit is installed on your host.
- Orbix is configured to run on your host platform.
- Your Java development kit (JDK) is configured to use the Orbix ORB runtime (see "Setting ORB Properties for the Orbix ORB" on page 11).

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

Setting ORB Properties for the Orbix ORB

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

- Using the `orb.properties` file.
- Using Java interpreter arguments.

Using the orb.properties File

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

Location of the orb.properties File.

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

Contents of the orb.properties File.

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

```
org.omg.CORBA.ORBClass=IE.Iona.OrbixWeb.CORBA.ORB
org.omg.CORBA.ORBSingletonClass=IE.Iona.OrbixWeb.CORBA.singleton
   ORB
```

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

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

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

Using Java Interpreter Arguments

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

```
java -Dorg.omg.CORBA.ORB=IE.Iona.OrbixWeb.CORBA.ORB
   -Dorg.omg.CORBA.ORBSingletonClass=IE.Iona.OrbixWeb.CORBA
   .singletonORB orbix_app
```

Developing Applications with Orbix Java

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

This chapter illustrates the programming steps using a banking example. In this example, an Orbix Java server program implements two types of objects: a single object implementing the `Bank` interface, and multiple objects implementing the `Account` interface. A client program uses these clearly defined object interfaces to create and find accounts, and to deposit and withdraw money.

The source code for the example described in this chapter is available in the `demos\BankSimpleTie` directory of your Orbix Java installation.

Developing a Distributed Application with Orbix Java

To create a distributed client-server application in Java using Orbix Java, you must perform the following programming steps:

1. Define the IDL interfaces.
2. Compile the IDL interfaces.
3. Implement the IDL interfaces.
4. Write the server application.
5. Write the client application.
6. Compile the client and server.
7. Register the server in the Implementation Repository.
8. Run the client.

This chapter outlines these programming steps in detail, using a banking example.

Defining IDL Interfaces

Defining IDL interfaces to your objects is the most important step in developing an Orbix Java application. These interfaces define how clients access objects, regardless of the location of those objects on the network.

An interface definition contains *attributes* and *operations*. Attributes allow clients to read and write to values on an object. Operations are functions that clients can call on an object.

For example, the following IDL from the banking example defines two interfaces for objects that represent a bank application. These interfaces are defined within an IDL module to prevent clashes with similarly named interfaces defined in subsequent examples.

The IDL interfaces to the banking example are defined as follows:

```
        // IDL
        // In file Banksimple.idl

1      module BankSimpleTie {
           typedef float CashAmount;

2              interface Account;

3                  interface Bank {
                   Account create_account (in string name);
                   Account find_account (in string name);
               };

4                  interface Account {
                   readonly attribute string name;
                   readonly attribute CashAmount balance;

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

This code is explained as follows:

1. An IDL module is a container construct that groups IDL definitions into a common namespace. Using a module is not mandatory, but it is good practice.
2. This is a forward declaration to the `Account` interface. This allows you to refer to `Account` in the `Bank` interface, before actually defining `Account`.
3. The `Bank` interface contains two operations: `create_account()` and `find_account()`, allowing a client to create and search for an account.
4. The `Account` interface contains two *readonly* attributes: `name` and `balance`. Clients can read a balance or name, but cannot write to them. If the `readonly` keyword is omitted, clients can also write to these values.
5. The `Account` interface also contains two operations: `deposit()` and `withdraw()`. The `deposit()` operation allows a client to deposit money in an account. The `withdraw()` operation allows a client to withdraw money from an account.

The parameters to these operations are labelled with the IDL keyword `in`. This means that their values are passed from the client to the object. Operation parameters can be labelled as `in`, `out` (passed from the object to the client) or `inout` (passed in both directions).

Compiling IDL Interfaces

You must compile IDL definitions using the Orbix Java IDL compiler. Before running the IDL compiler, ensure that your configuration is correct.

Checking your Configuration

To set up configuration for the IDL compiler, you should check that Orbix Java can find its root configuration file, `iona.cfg`.

You should ensure that the environment variable `IT_CONFIG_PATH` is set to the directory in which `iona.cfg` resides. By default, this is the `config` directory of your installation. You should also include the `config` directory on your classpath.

Running the IDL Compiler

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

```
idlj -jP Demos BankSimple.idl
```

This command generates a number of Java files that are used to communicate with Orbix Java. The generated files are located in the `Demos\BankSimpleTie\java_output` directory. Discussion of these files is deferred until, “Orbix Java 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 banking example, all application files are placed within a package called `Demos.BankSimpleTie`.

Implementing IDL Interfaces

You must implement the IDL interfaces using the code generated by the IDL compiler. The banking example uses the *TIE approach* to implement its IDL interfaces. You can also use the *ImplBase approach*. Both of these approaches are discussed in detail in “Implementing the IDL Interfaces” on page 105.

Implementing the Bank Interface

Implementing the `Bank` IDL interface using the *TIE* approach involves creating an implementation class that implements the IDL-generated class `_BankOperations`.

In this example, the implementation class created for the `Bank` IDL interface is `BankImplementation`.

```
    // Java
    // In file BankImplementation.java

    package Demos.BankSimpleTie;

    import IE.Iona.OrbixWeb._OrbixWeb;
    import org.omg.CORBA.ORB;
    import org.omg.CORBA.SystemException;
    import java.util.*;
1   public class BankImplementation
    implements _BankOperations {

    // Default Constructor.
2   public BankImplementation ( org.omg.CORBA.ORB Orb ) {
        m_orb = Orb;
        m_list = new Hashtable();
```

```
      }

      // Implementation for create_account().
3     public Account create_account ( String name ) {
        Account m_account = null;
        AccountImplementation m_account_impl = null;

        // Check if account already exists.
        if ( m_list.get ( name ) != null ) {
          System.out.println ( "- Account for " + name
          + " already exists, " + "finding details." );
          return find_account ( name );
        }

        System.out.println ("Creating new account for "
            + name +".");

        // Create a new account.
        try {
4         m_account_impl = new AccountImplementation(name,
            0.0F);
          m_account = new _tie_Account(m_account_impl,
            “Marker”);
5         m_orb.connect(m_account);
        }
        catch ( SystemException se ) {
          System.out.println ( "[ Exception raised when
                   creating Account. ]" );
        }

        // Add account to table of accounts.
        m_list.put ( name, m_account );
        return m_account ;
      }

      // Implementation for find_account().
6     public Account find_account ( String name ) {
        Account m_acc = null;
        m_acc = ( Account ) m_list.get ( name );

        if ( m_acc == null ) {
          // Account not in table.
          System.out.println ("Unable to find Account for"
              + name + "." );
        }
        return m_acc;
      }
    ...
    }
```

This code is described as follows:

1. The implementation class must implement the IDL-generated interface _BankOperations. This maps the attributes and operations in the IDL definitions to Java methods.
2. The Orb parameter to the BankImplementation default constructor refers to the server's ORB.
3. The implementation for the IDL operation create_account() takes the account name as a parameter and returns a reference to the newly created account.
4. Using the TIE approach, you must *tie together* the implementation class and the IDL interface using the automatically generated Java TIE class.

   In this example, the IDL compiler generates the TIE class _tie_Account for the IDL interface Account. You must then pass an object that implements the IDL interface as a parameter to the constructor for the TIE class.
5. Connect the implementation object to the Orbix Java runtime.
6. The implementation for the IDL operation find_account() takes the account name as a parameter and returns a reference to the account searched for.

Implementing the Account Interface

The implementation class for the Account IDL interface should inherit from the IDL-generated interface _AccountOperations:

```
// Java
// In file BankImplementation.java

package Demos.BankSimpleTie;

public class AccountImplementation
    implements _AccountOperations {

    // Constructor
    public AccountImplementation(String name,float bal){
        this.m_name = name;
        m_balance=bal;
        System.out.println ("- Creating account for " +
            m_name + ". Initial " + "balance of £" + bal );
    }

    // Implementation for IDL name accessor.
    public String name() {
        return m_name;
    }
    // Implementation for IDL balance accessor.
    public float balance() {
        return m_balance;
    }

    // Implementation for IDL operation deposit().
    public void deposit ( float amount ){
        System.out.println ( "- Depositing £" + amount + "
            into " + m_name + "'s account" );
        m_balance += amount;
    }

    // Implementation for IDL operation withdraw().
    public void withdraw ( float amount ) {
        System.out.println ( "- Withdrawing £" + amount +
            " from " + m_name + "'s account" );
            m_balance -= amount;
    }
...
}
```

The IDL attributes name and balance are implemented by corresponding Java accessor methods. All mapped attributes and operations are defined in the Java interface _AccountOperations, generated by the IDL compiler.

Writing an Orbix Java Server Application

To write a Java program that acts as an Orbix server, perform the following steps:

1. Initialize the server connection to the ORB.
2. Create an implementation object by creating instances of the implementation classes.
3. Register the implementation object in the Naming Service.

This section describes each of these programming steps in turn.

Initializing the ORB

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 CORBA standard can then be invoked on this instance. You should use the parameterized version of the `init()` method, defined as follows:

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

This method is passed an array of strings as command-line 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 ***Orbix Programmer's Reference Java Edition*** for further details on the `org.omg.CORBA.ORB` class.

Creating an Implementation Object

To create an implementation object, you must create an instance of your implementation class in your server program. Typically a server program creates a small number of objects in its `main()` function, and these objects may in turn create further objects. In the banking example, the server creates a single `Bank` object in its `main()` function. This bank object then creates accounts when `create_account()` is called by the client.

For example, to create an instance of the `Bank` IDL interface in your server `main()` function, using the TIE approach, do the following:

```
Bank m_bank = new _tie_Bank(new BankImplementation(m_orb),
“myBankMarker”);
```

This creates a new server implementation object, passing a reference to the server ORB.

Registering an Object with the Naming Service

You must register your implementation objects with the CORBA Naming Service. This provides a flexible CORBA-defined way to locate objects. The Naming Service allows a name to be *bound* to an object, and allows that object to be found subsequently by *resolving* that name within the Naming Service.

CORBA Object References

A CORBA *object reference* identifies an object in your system. A server that holds an object reference can register it with the Naming Service, giving it a name that can be used by other components of the system to find the object.

The Naming Service maintains a database of *bindings* between names and object references. A binding is an association between a name and an object reference. Clients can call the Naming Service to resolve a name, and this returns the object reference bound to that name.

The Naming Service provides operations to resolve a name, to create new bindings, to delete existing bindings, and to list the bound names. A name is always resolved within a given naming context.

The following server program initializes the ORB, creates a BankImplementation object, and registers this object in the Naming Service:

```
    // Java
    // In file Server.java.

    package Demos.BankSimpleTie;

1   import Demos.IT_DemoLib.*;
    import IE.Iona.OrbixWeb.Features.Config;
    import IE.Iona.OrbixWeb._OrbixWeb;
    import org.omg.CORBA.*;

    public class Server {

        public static void main ( String args[] ) {

            // Initalize the ORB
2           org.omg.CORBA.ORB Orb = ORB.init ( args, null );


            // Create a new bank Server
            new Server (Orb);
        }

        // Server constructor.
3       public Server ( org.omg.CORBA.ORB Orb ) {
            m_orb = Orb;
            System.out.println( "Server started on port "
             + Config.getConfigItem ("IT_IIOP_LISTEN_PORT")
    );

            // Create a new Naming Service wrapper.
            try {
4              m_ns_wrapper = new IT_NS_Wrapper(m_orb,
                m_demo_context_name);
                m_ns_wrapper.initialise();
            }
            catch ( org.omg.CORBA.UserException userEx ) {
            System.out.println ( "[ Exception raised during
                creation of naming "+ "service wrapper.]" );
            }

            String serverName = new String ("IT_Demo/
                                    BankSimple/Bank");
5          m_bank = new _tie_Bank ( new BankImplementation
                                 (m_orb));
            try {
6          m_ns_wrapper.registerObject ("Bank", m_bank );
            }
            catch ( org.omg.CORBA.UserException userEx ) {
                System.out.println( "[Exception registering
                                     Bank in " +
    "NamingService.]");
            }

            // Wait for client connections.
            try {
7             _OrbixWeb.ORB (m_orb).processEvents
```

```
                                    ( 10000 * 60 );
            }
            catch ( SystemException se ) {
            System.out.println("[ Exception during creation
            of implementation : " + se.toString() + " ]" );
                System.exit(1);
            }
        }
        ...
        private final String m_demo_context_name =
                                    “IT_Demo.BankSimple";
    }
```

This code is described as follows:

1. To simplify the use of the Naming Service, a Naming Service wrapper is provided. This hides the low-level detail of the CORBA Naming Service.
2. Initialize the ORB for an Orbix Java application using the parameterized version of `ORB.init()`.
3. The `Server()` constructor creates the bank implementation object and adds an entry for it to the Naming Service. If the entry already exists, it is replaced. The `Orb` parameter refers to the server’s ORB.
4. Create a Naming Service wrapper object. The banking example uses Naming Service wrapper methods to simplify the use of the Naming Service.
5. Create a new server implementation object, and pass a reference to the server ORB.
6. Register the bank object in the Naming Service using the wrapper method `registerObject()`. This object is now known as `Bank` in the Naming Service.
7. Start the server listening for incoming invocations.

For details of different methods for connecting the implementation objects to the Orbix Java runtime, refer to “Object Initialization and Connection” on page 113.

Error Handling for Server Applications

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

In the banking example, the code in the `catch` clause displays details of possible system exceptions raised by Orbix Java. It does this by printing the result of the `SystemException.toString()` method to the Java `System.out` print stream.

The constructor for the IDL-generated `_BankOperations` type may raise a system exception, so the instantiation of this object must be enclosed in a `try` statement. Refer to “Exception Handling” on page 143 for more details.

Writing the Client Application

Writing client applications involves writing Java clients that access implementation objects through IDL interfaces. You must perform the following steps:

1. Initialize the client connection to the ORB.
2. Get a reference to an object.
3. Invoke attributes and operations defined in the object's IDL interface.

This section describes each of these steps in turn.

Initializing the ORB

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 parameterized version of the `init()` method, defined as follows:

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

In the banking example, the client initializes the ORB, and passes it as a parameter to the client constructor, as follows:

```
// Java
// In file Client.java

    // Initilize the ORB
    org.omg.CORBA.ORB Orb = ORB.init ( args,null );

    // Create a new client
    new Client (Orb);
```

Getting a Reference to an Object

The CORBA-defined way to get a reference to an object is to use the Naming Service. When an object reference enters a client address space, Orbix Java creates a *proxy* object that acts as a local representative for the remote implementation object. Orbix Java forwards operation invocations on the proxy object to corresponding methods in the implementation object.

The following sample code shows how the client uses Naming Service wrapper methods to obtain an object reference:

```
    // Java
    // In file Client.java
    ...

    public void connectToBank {
        // Get the hostname from the user interface.
1       String host = m_client_frame.Get_HostName();

        _OrbixWeb.ORB(m_orb).setConfigItem
                    ("IT_NAMES_SERVER_HOST",host );

        try {
2           m_ns_wrapper = new IT_NS_Wrapper ( m_orb,
                m_demo_context_name);
        }
        catch ( org.omg.CORBA.UserException userEx ) {
        m_client_frame.printToMessageWindow ("[ Exception
            raised during creation of naming" + "service
            wrapper.]" );
        }
        try {
3               org.omg.CORBA.Object obj = m_ns_wrapper.resolveName
                                                    ("Bank");
4               m_bank = BankHelper.narrow ( obj );
            m_client_frame.printToMessageWindow("Connection
                                        succeeded." );
        }
        catch ( org.omg.CORBA.UserException userEx ) {
        m_client_frame.printToMessageWindow ( "[ Exception
        raised getting Bank reference " + userEx + "]" );
        }
    }
```

This code is described as follows:

1. Set the Naming Service hostname to the hostname input by the user.
2. Create a new Naming Service wrapper object.
3. The method `m_ns_wrapper.resolveName()` retrieves the object reference from the Naming Service placed there by the server. The parameter is the name of the object to resolve. This must match the name used by the server when it called `registerObject()`.
4. The return type from `resolveName()` is of type `org.omg.CORBA.Object`. You must call `BankHelper.narrow()` to cast from this base class to the `Bank` IDL class, before you can make invocations on remote `Bank` objects. The client stub code generated for every IDL class contains the `BankHelper.narrow()` method definition for that class.

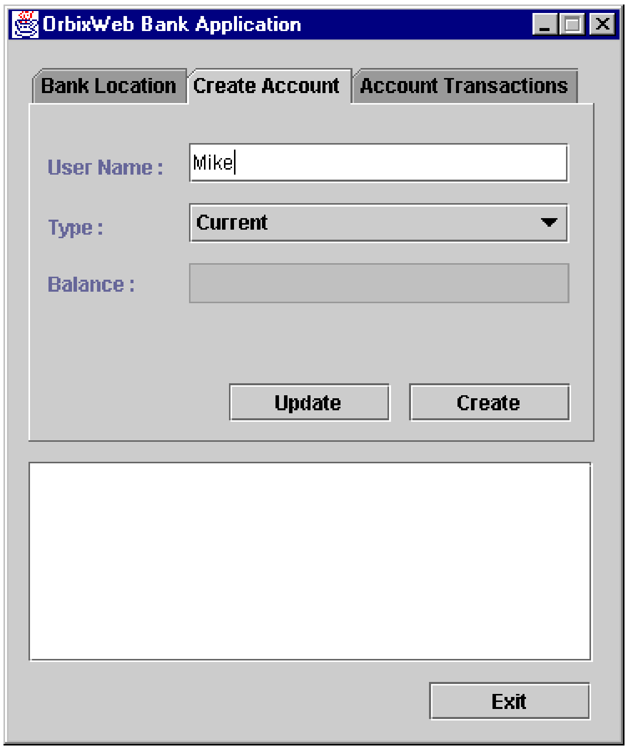
Invoking IDL Attributes and Operations

To access an attribute or an operation associated with an object, call the appropriate Java method on the object reference. The client-side proxy redirects this call across the network to the appropriate Java method for the implementation object.

Orbix Java enables you to invoke IDL operations using normal Java method calls. The following code extract shows the code called when you choose to create an account, using the interactive GUI shown in Figure 6 on page 24:

```
    // Create a user account.
public void createNewAccount() {
    Account new_account = null;
    String current_name = m_client_frame.Get_UserName();
    try {
        // Call the IDL-defined method create_account().
        new_account = m_bank.create_account(current_name);
        m_client_frame.Set_Balance(0f);
        m_client_frame.printToMessageWindow("Created
            account for " + current_name + "." );
    }
```

```
        catch ( SystemException se ) {
            m_client_frame.printToMessageWindow ( "[ Exception
                raised during account creation. " + se + " ] ");
        }
    }
```



User Name : Mike

Type : Current

Balance :

Figure 6: *Creating an Account using the Bank GUI*

Similarly, the following Java code is called when you choose to find an account:

```
// Java.
// In file Client.java.
...
// Find a user account.
private Account getCurrentUserAccount() {
    try {
        // Call the IDL-defined method find_account().
        return m_bank.find_account
            (m_client_frame.Get_UserName() );
    }
    catch ( SystemException ex ) {
        m_client_frame.printToMessageWindow
            ( "[ Exception raised finding account for "
                + m_client_frame.Get_UserName()+ "]");
    }
    return null;
}
```

The following code extract shows the Java code called when the user chooses to make a deposit into an account:

```
// Java.
// In file ClientGUIFrame.java.

public void DepositButton_mouseClicked() {
    Account user_account =
            m_client.getCurrentUserAccount();

    if ( user_account != null ) {
        try {
            user_account.deposit
                (Get_Transaction_Amount() );
            m_client.updateCurrentUserBalance();
        }
        catch ( SystemException se ) {
            printToMessageWindow ( "[ Exception raised
                        during account deposit. ]" );
        }
    }
}
```

Compiling the Client and Server

Details of compiling the client and server are specific to the Java development environment used. However, it is possible to describe general requirements. These are illustrated here using the Oracle Java Developer's Kit (JDK), version 1.1.x or higher. This is the development environment used by the Orbix Java demonstration makefiles.

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

- The Java API classes located in the `rt.jar` file in the `jre/lib` directory of your JDK installation.
- The Orbix Java API classes located in the `OrbixWeb.jar` file in the `lib` directory of your Orbix Java installation.
- The `config` directory of your Orbix Java installation.
- Any other 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 files generated by the IDL compiler. In the banking server example, the user-generated source files are as follows:

- `Server.java`
- `BankImplementation.java`
- `AccountImplementation.java`

The IDL-generated files are as follows:

- `_BankSkeleton.java`
- `_BankOperations.java`

- `_tie_Bank.java`
- `_BankStub.java`
- `Bank.java`
- `_AccountSkeleton.java`
- `_AccountOperations.java`
- `_tie_Account.java`
- `_AccountStub.java`
- `Account.java`

The IDL-generated files are located in the `demos\BankSimpleTie\java` directory. Discussion of the IDL-generated files is deferred until "Orbix Java IDL Compilation" on page 29.

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 this example, the source file is `Client.java`, and the generated files are as follows:

- `_BankStub.java`
- `Bank.java`
- `_AccountStub.java`
- `Account.java`

Using Orbix Java Utilities

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

The Orbix Java `demos\BankSimpleTie` directory provides a script that calls `owjavac.pl` as required. To compile the Java source files, enter the appropriate command from the `BankSimpleTie\java` directory:

UNIX	`% make`
Windows	`> compile`

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

For details on the use of the `owjava.pl` and `owjavac.pl` wrapper utilities, refer to "Using the Orbix Java Wrapper Utilities" on page 186.

Registering the Server

Registering the server in the Implementation Repository 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 `Bank` object.

Running the Orbix Java Daemon

Before registering the server, you should ensure that an Orbix C++ or Java daemon process (`orbixd` or `orbixdj`) is running on the server machine.

To run the Orbix Java daemon, enter the `orbixdj` command from the `bin` directory of your Orbix Java installation. To run the Orbix C++ or Java daemon, enter the `orbixd` command.

On Windows, you can also start a daemon process by clicking on the appropriate menu item from the Orbix Java folder.

Using Putitj

Once an Orbix Java daemon process is running, you can register the server. To register the `Bank` server, use the `putitj` command as follows:

```
putitj -j Bank Demos.BankSimpleTie.Server
```

The `-j` switch indicates that the specified server should be launched via the Java Interpreter. The second parameter to `putitj` is the server name, `Bank` in this example. The third parameter is the name of the class that contains the server's `main()` method (`Demos.BankSimpleTie.Server` in this example). This is the class that should be run through the Java interpreter.

The server registration step is automated by a script in the `demos\BankSimpleTie` directory that executes the `putitj` command. Refer to the ***Orbix Administrator's Guide Java Edition*** for more details on the `putitj` command.

Running the Client Application

To run the client application you must run the Java interpreter on the bytecode (`.class` files) produced by the Java compiler. When running an Orbix Java client application, you must ensure that the interpreter can access the following:

- The Java API classes located in the `rt.jar` file in the `jre/lib` directory of your JDK installation.
- The Orbix Java API classes located in the `OrbixWeb.jar` file in the `lib` directory of your Orbix Java installation.
- The `config` directory of your Orbix Java installation.
- Any other classes required by the application.

Using Orbix Java Utilities

You can use the owjava.pl 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.pl tool passes the default classpath to the interpreter, avoiding the need to set up environment variables. Refer to "Using the Orbix Java Wrapper Utilities" on page 186 for more details on this convenience tool.

A script named Client in the demos\BankSimpleTie\java directory implements this step. To run the client application, use the following command:

```
Client server_host
```

The Bank GUI then appears as shown in Figure 7 on page 28.

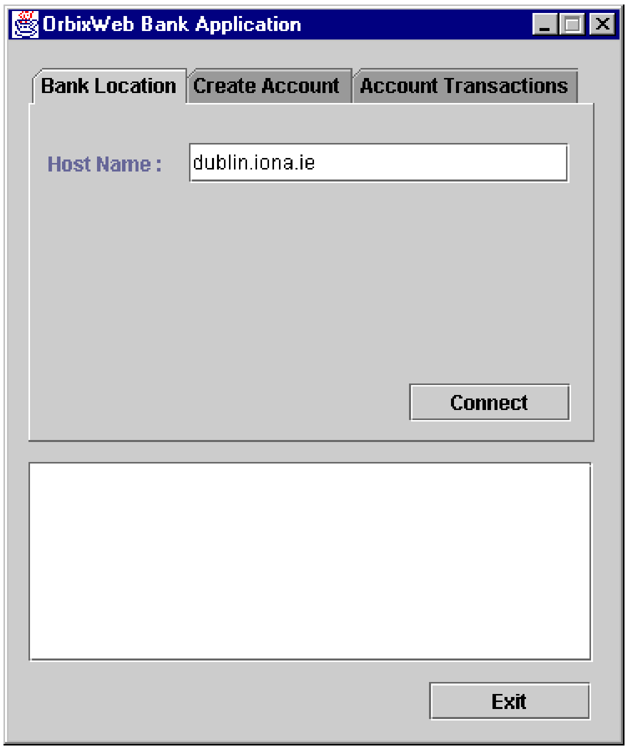


Figure 7: *The Bank GUI.*

Summary of the Programming Steps

The steps involved in creating a distributed client-server application using Orbix Java are as follows:

1. Define the interfaces to objects used by the application, using 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 as follows:
    i. Initialize the server connection to the ORB.
    ii. Create an implementation object by creating instances of the implementation classes.
    iii. Register the implementation object in the Naming Service.
5. Write a client application to use the CORBA objects located in the server as follows:
    i. Initialize the client connection to the ORB.
    ii. Get a *reference* to an object.
    iii. Invoke attributes and operations defined in the object's IDL interface.
6. Compile the client and server applications.
7. Register the server in the Orbix Java Implementation Repository.
8. Run the client application.

Orbix Java IDL Compilation

This section examines the Orbix Java IDL compilation process, focusing on the Java classes and interfaces generated by the IDL compiler.

The Orbix Java IDL compiler produces Java code corresponding to the IDL definitions. For example, the mapped Java code consists of code that allows a client to access an object through the `Bank` interface, and code that allows a `Bank` object to be implemented in a server.

The IDL compilation produces Java constructs (six classes and two interfaces) from the IDL interface `Bank`. Each public Java 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 Orbix Java IDL compiler creates a local `java_output` directory into which the generated Java directory structure is placed. You can specify an alternative target directory using the compiler switch "-jO directory".

Each generated file contains a Java class or interface that serves a specific role in an application. For example, the following files are generated for the Bank IDL interface:

Client-Side Mapping	Description
Bank	A Java interface whose methods define the Java client view of the IDL interface.
_BankStub	A Java class that implements the methods defined in the Bank interface. This class provides functionality that allows client method calls to be forwarded to a server.
Server-Side Mapping	**Description**
_BankSkeleton	A Java class used internally by Orbix Java to forward incoming server requests to implementation objects. You do not need to know the details of this class.
_BankImplBase	An abstract Java class that allows server-side developers to implement the Bank interface using the ImplBase approach.
_tie_Bank	A Java class that allows server-side developers to implement the Bank interface using delegation. This approach to interface implementation is called the TIE approach. The TIE approach is an Orbix Java -specific feature, and is not defined by the CORBA specification. It is the recommended approach for Orbix Java due to the restriction to single inheritance in Java.
_BankOperations	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, using the TIE approach.
Client and Server-Side Mapping	**Description**
BankHelper	A Java class that allows you to manipulate IDL user-defined types in various ways.
BankHolder	A Java class defining a Holder type for class Bank. This is required for passing Bank objects as inout or out parameters to and from IDL operations. Refer to "Holder Classes and Parameter Passing".
BankPackage	A Java package used to contain any IDL types nested within the Bank interface; for example, structures or unions.

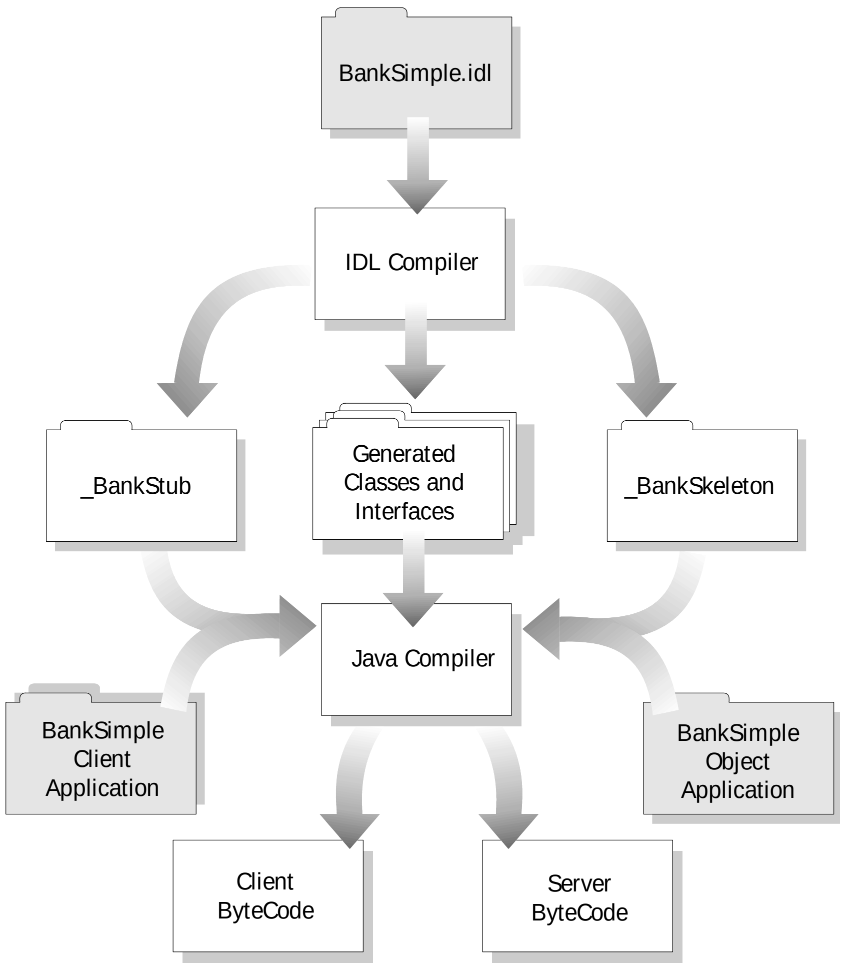


Figure 8: *Overview of the Compiling the Bank IDL Interface*

Examining 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.

Client-Side Mapping

The Java files Bank.java and _BankStub.java support the client-side mapping. The Bank.java file maps the operations and attributes in BankSimple.idl to Java methods as follows:

```
// Generated by the Orbix Java IDL compiler

package Demos.BankSimpleTie;

public interface Bank extends org.omg.CORBA.Object {
    public Demos.BankSimpleTie.Account create_account
                                 (String name) ;
    public Demos.BankSimpleTie.Account find_account
                                 (String name) ;
    public java.lang.Object _deref() ;
}
```

This Java interface defines an Orbix Java client view of the IDL interface defined in BankSimple.idl. The Java interface is implemented by the Java class _BankStub in the file _BankStub.java as follows:

```
// Generated by the Orbix Java IDL compiler

package Demos.BankSimpleTie;

public class _BankStub
    extends org.omg.CORBA.portable.ObjectImpl
    implements Demos.BankSimpleTie.Bank {

    public _BankStub () {}

    public Demos.BankSimpleTie.Account
        create_account(String name) {
    ...
    }
    public Demos.BankSimpleTie.Account
        find_account(String name) {
    ...
    }

    public static final String _interfaces[] =
                   {"IDL:BankSimple/Bank:1.0"};

    public String[] _ids() {return _interfaces;}

    public java.lang.Object _deref() {return null;}

}
```

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

The `create_account()` and `find_account()` 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" for a complete description.

Server-Side Mapping

Orbix Java provides support for two approaches to implementing an IDL interface:

- The *TIE* approach, which uses delegation.

  The generated Java constructs used in the TIE approach are the interface `_BankOperations` and the class `_tie_Bank`.

  The TIE approach is used in this chapter to implement the `BankSimple` IDL interfaces.
- The *ImplBase* approach, which uses inheritance.

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

The use of the TIE and ImplBase approaches is discussed in detail in "Implementing the IDL Interfaces" on page 105. The TIE approach, which uses delegation, is preferred for many Java applications and applets.

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.

Developing Applets with Orbix Java

This chapter extends the banking example from "Developing Applications with Orbix Java". It explains how to use Orbix Java to create a downloadable client applet that communicates with a back-end server. The programming steps differ on the client side only. You should be familiar with the material covered in "Developing Applications with Orbix Java" before continuing with this chapter.

Review of Orbix Java Programming Steps

Recall the programming steps typically required to create a distributed client-server application using Orbix Java:

1. Define the interfaces to objects used by the application, using CORBA 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 Orbix Java 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 banking IDL interface outlined in "Defining IDL Interfaces" on page 13. The sample code described in this chapter is available in the `demos/common/BankSimpleApplet` directory of your Orbix Java installation.

Providing a Server

This chapter illustrates a distributed architecture in which a downloadable client applet communicates with an Orbix Java server through an IDL interface. This client-server architecture is a common requirement in the Java environment where small, dynamic client applets are downloaded to communicate with large, powerful back-end service applications. Architectures in which full Orbix Java 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 an Orbix Java Server Application" on page 18. The Orbix Java programming steps for writing servers are identical for Java applications and Java applets. The main differences between programming for Java applications and Java applets occur when writing the client.

Writing a Client Applet

This section develops a simple Java applet, providing a graphical user interface to the banking IDL interface. The example used builds upon the concepts already introduced in "Writing the Client Application" on page 22.

Writing the client applet can be broken down into four sub-steps, each corresponding to a particular demonstration source file, as follows:

Programming Step	**Source File**
1. Creating the user interface	`BankPanel.java`
2. Adding Orbix Java client functionality	`BankEvents.java`
3. Creating the applet	`BankApplet.java`
4. Adding the client to a HTML file	`Index.html`

These files are located in the `demos\common\BankSimpleApplet\java` directory of your Orbix Java installation. The package name for the Java classes in this example is `Demos.BankSimpleApplet`. This example assumes that the file `BankSimple.idl` is compiled with the following command:

```
    idlj -jP Demos BankSimple.idl
```

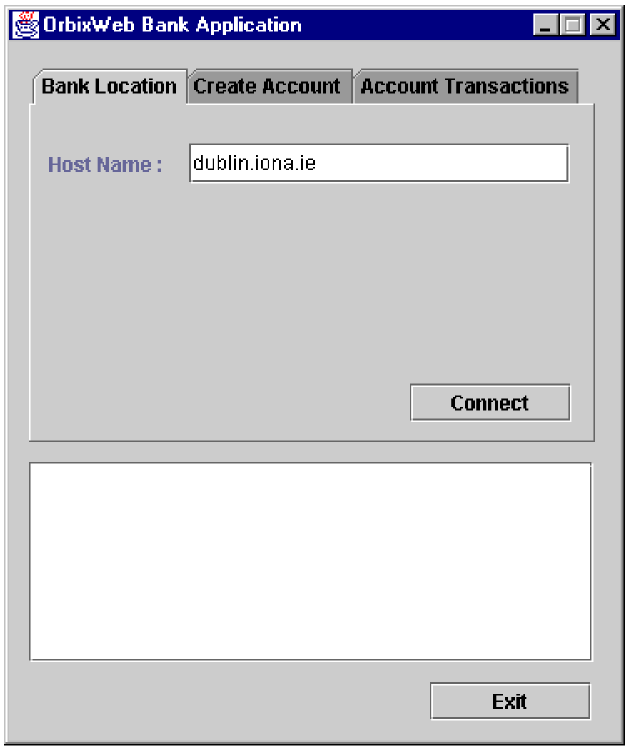
Developing an Orbix Java client can be completely decoupled from developing the server. For this reason, when compiling the IDL file, the package name chosen for the client can differ from the package name for the server.

Creating the User Interface

The GUI source code in `BankPanel.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 BankSimpleApplet GUI shown in Figure 9 on page 37 consists of three tabs:

Bank Location	Used to specify a Naming Service host and get a reference to a `Bank` object.
Accounts	Used to create, find and update specified accounts.
Transactions	Used to make withdrawals and deposits for specified accounts.



Host Name : dublin.iona.ie

Figure 9: *The Banking Graphical User Interface*

The following code sample names the individual GUI components, such as buttons and text fields. The details of how the GUI is implemented are not discussed:

```
// Java
// In file BankPanel.java.

package Demos.BankSimpleApplet;
import java.awt.*;

public class BankPanel extends Panel {

    // Button String constants
    final String m_connect_string = "Connect";
    final String m_disconnect_string = "Disconnect";
    final String m_withdraw_string = "Withdraw";
    final String m_deposit_string = "Deposit";
    final String m_create_string = "Create New Account";
    final String m_update_string = "Update";

    // Labels
    Label m_user_label = new Label ("Username");
    Label m_balance_label = new Label ("Acount Balance");
    Label m_transaction_label = new Label
                               ("Transaction Amount");
    Label m_hostname_label = new Label ("Host");
    // Buttons
    Button m_connect_button;
    Button m_disconnect_button;
    Button m_withdraw_button;
    Button m_deposit_button;
    Button m_create_button;
    Button m_update_button;

    // Text fields
    TextField m_transaction_field;
    TextField m_user_field;
    TextField m_balance_field;
    TextField m_hostname_field;

    // Sub panels
    Panel m_top_panel = new Panel();
    Panel m_bottom_panel = new Panel();
    // Constructor
    public BankPanel() {
    ...
    }
...
}
```

Adding Orbix Java Client Functionality

In the banking applet example, all Orbix Java 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 Bank 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 Bank or Account proxy object.

A subclass of BankPanel named BankEvents acts as the container for the various buttons and text fields. The following is an outline of the source code for the class BankEvents. The button implementation methods defined here are expanded on later in this section:

```
// Java
// In file BankEvents.java.

package Demos.BankSimpleApplet;

import java.awt.*;
import Demos.IT_DemoLib.*;
import org.omg.CORBA.ORB;
import org.omg.CORBA.SystemException;
import IE.Iona.OrbixWeb._OrbixWeb;
import IE.Iona.OrbixWeb._CORBA;

public class BankEvents extends BankPanel {

    // Constructor.
    public BankEvents(){
    super();
    org.omg.CORBA.ORB Orb = ORB.init(this,null);
    m_orb = Orb;
  }

    // Notify appropriate method for action event.
  public boolean action (Event event, Object arg) {
    if ( m_connect_string.equals(arg)){
      connect();
    }

        else if ( m_disconnect_string.equals(arg)){
            disconnect();
        }
        else if ( m_withdraw_string.equals(arg)){
            withdraw();
        }
        else if ( m_deposit_string.equals(arg)) {
            deposit();
        }
        else if (m_create_string.equals(arg)) {
            create();
        }
        else if (m_update_string.equals(arg)) {
            update();
        }
        return true;
        }
        // Connect button implementation.
        public void connect() {
            // Details later in this section.
        }

        // Exit button implementation.
        public void disconnect() {
            m_bank = null;
        }

        public void update() {
            updateCurrentUserBalance();
        }

        // Deposit button implementation.
        public void deposit() {
            // Details later in this section.
        }

        // Withdraw button implementation.
        public void withdraw() {
            ...
        }



        // Create button implementation.
        public void create() {
        // Details later in this section.
        }

        // Update button implementation.
        private void updateCurrentUserBalance() {
            ...
        }

        // Find button implementation.
        private Account getCurrentUserAccount() {
        // Details later in this section.
        }

        private void displayMsg (String msg) {
        // Details later in this section.
        }
        ...
    }
```

The BankEvents class provides methods to handle the client functionality required for the GUI buttons shown in Figure 9 on page 37. The following sections explain the button implementations in detail.

Getting a Reference to an Object

The CORBA-defined way to get a reference to an object is to use the Naming Service. When an object reference enters a client address space, Orbix Java creates a *proxy* object that acts as a local representative for the remote implementation object. Orbix Java forwards operation invocations on the proxy object to corresponding methods in the implementation object.

The **Connect** button on the **Bank Location** tab is implemented by the Connect() method. This uses Naming Service wrapper functions to obtain a Bank object reference:

```
    public void connect() {
        String hostname;

        // Get hostname from the text field.
        hostname = m_hostname_field.getText();
        try {
            //Set the naming service hostname
1           _OrbixWeb.ORB (m_orb).setConfigItem
                    ("IT_NAMES_SERVER_HOST", hostname);
            }
        catch(Exception ex){
            displayMsg("First exception caught:
                                "+ex.toString());
        }

        // Create a new Naming Service wrapper
        try {
2        m_ns_wrapper = new IT_NS_Wrapper
                            (m_orb, m_demo_context_name);
        }
        catch (org.omg.CORBA.UserException user_ex) {
            displayMsg("Exception raised during creation of
            naming service wrapper: " + user_ex.toString());
        }
        try {
                org.omg.CORBA.Object m_obj =
3             m_ns_wrapper.resolveName("Bank");
                displayMsg("After resolving name");

4 m_bank = BankHelper.narrow(m_obj);
                displayMsg("Connect succeeded!");
            }
            catch(org.omg.CORBA.UserException user_ex) {
                displayMsg("Exception raised getting bank
                        reference: "+user_ex.toString());
            }
            catch (Exception ex) {
             displayMsg("Exception caught: "+ex.toString());
            }
        }
    }
```

This code is described as follows:

1. Set the Naming Service hostname to that input by the user.
2. Create a new Naming Service wrapper object.

3. The method `nsWrapper.resolveName()` retrieves the object reference from the Naming Service placed there by the server. The parameter is the name of the object to resolve, in this case `Bank`.This must match the name used by the server when it called `registerObject()`.
4. The return type from `resolveName()` is of type `org.omg.CORBA.Object`. You must call `BankHelper.narrow()` to cast from this base class to the `Bank` IDL class, before you can make invocations on remote `Bank` objects. The client stub code generated for every IDL class contains the `BankHelper.narrow()` function definition for that class.

Disconnecting from a Server

The **Exit** button functionality is implemented as follows:

```
public void disconnect() {
    m_bank = null;
}
```

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

```
m_orb.closeConnection(m_bank);
```

Invoking IDL Attributes and Operations

To access an attribute or an operation associated with an object, call the appropriate Java method on the object reference. The client-side proxy redirects this call across the network to the appropriate Java method for the implementation object.

Orbix Java enables you to invoke IDL operations using normal Java method calls. The following code extracts show the code called when you select the appropriate GUI button.

Creating an Account

The **Create** button functionality is implemented as follows:

```
// Create button implementation.
public void create() {
    Account new_account = null;
    String current_name = m_user_field.getText();
    try{
        new_account = m_bank.create_account
                            (current_name);
        m_balance_field.setText
            (String.valueOf((float)0));
        displayMsg("Created an account for "+
                            current_name);
    }
    catch (SystemException se) {
        displayMsg("Exception raised during creation
                        of account "+se.toString());
    }
}
```

The `create()` method enables the IDL-defined method `create_account()` to be called on the proxy object `m_bank`.

Finding an Account

The **Find** button functionality is implemented as follows:

```
// Find button implementation.
private Account getCurrentUserAccount() {
    try {
        return m_bank.find_account
                (m_user_field.getText());
    }
    catch(SystemException se){
        displayMsg("Exception raised finding account
                    for "+ m_user_field.getText());
    }
    return null;
}
```

This enables the IDL-defined method `find_account()` to be called on the `m_bank` proxy object.

Making a Deposit

The **Deposit** button functionality is implemented as follows:

```
public void deposit() {
    Account user_account = getCurrentUserAccount();
    float amount = Float.valueOf
    (m_transaction_field.getText()).floatValue();

    if (user_account != null) {
        try {
            user_account.deposit(amount);
            updateCurrentUserBalance();
        }

        catch (SystemException se) {
            displayMsg("Exception raised while
                attempting a deposit "+se.toString());
        }
    }
}
```

This allows the IDL-defined `deposit()` method to be called on proxy objects located via the `find_account()` method. The **Withdraw** button functionality is implemented in a similar way.

Handling Exceptions in Orbix Java Client Applets

In the example described in "Writing the Client Application" on page 22, Orbix Java 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 Orbix Java 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 MsgDialog.java.
package Demos.BankSimpleApplet;

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. Orbix Java error-handling can be added to the BankEvents class by defining a display method as follows:

```
private void displayMsg (String msg) {
    Demos.SimpleBankApplet.MsgDialog m_msg_dialog =
        new Demos.SimpleBankApplet.MsgDialog
                ("Bank Operation Result", msg);
        m_msg_dialog.resize(380,200);
        m_msg_dialog.show();
}
```

This allows any string, including system exception strings, to be displayed in a dialog box.

Creating the Applet

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

```
// Java
// In file BankApplet.java.
package Demos.BankSimpleApplet;

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

public class BankApplet extends Applet {

    // Main display panel
    BankEvents m_bank_events;

    public void init () {
    try {
        ORB.init(this, null);
    }
    catch (INITIALIZE ex) {
        System.err.println ("failed to initialize: "+ex);
    }

    // Create new panel.
    m_bank_events = new BankEvents ();

    // Add panel to applet.
    this.add (m_bank_events);
    }
}
```

Initializing the ORB

Because Orbix Java 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, `BankApplet.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 *Orbix Programmer's Reference Java Edition* for further information on class `org.omg.CORBA.ORB` and `ORB.init()`.

Adding the Applet to a HTML File

In HTML terms, an Orbix Java applet client behaves exactly like a standard Java applet. It can be included in a HTML file using the standard `<APPLET>` tag, as shown in the file `Index.html`:

```
    // HTML
    // In file Index.html

    <HTML>
    <HEAD>
        <TITLE>Orbix Java BankSimpleApplet demo</TITLE>
    </HEAD>

    <BODY>
        <H1>Bank Client</H1>

        <APPLET CODE="Demos/BankSimpleApplet/
            BankSimpleApplet.class"
1           CODEBASE="../../classes/"
            archive="OrbixWeb.jar"
            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.singletonORB>
        </APPLET>
    </BODY>
    </HTML>
```

This HTML is described as follows:

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 Orbix Java ORB implementation. This means that Orbix Java -specific methods such as `bind()` can be used.

Compiling the Client Applet

The instructions for compiling an Orbix Java applet are identical to those for a standard Orbix Java application, as described in "Compiling the Client and Server" on page 25.

You must ensure that the Java compiler can access the Java API packages (including `java.awt` for this sample code), the Orbix Java `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 banking example:

- `_BankStub.java`
- `Bank.java`
- `_AccountStub.java`
- `Account.java`
- `BankPanel.java`
- `BankEvents.java`
- `BankApplet.java`
- `MsgDialog.java`
- `Msg.java`

The Orbix Java `demos/common/BankSimpleApplet` directory provides a script that invokes the `owjavac.pl` wrapper utility as required. To compile the client applet, enter the appropriate command at the operating system prompt:

UNIX	`% make`
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 Index.html
```

Java applets differ slightly from standalone Java applications in their requirements for accessing 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 Orbix Java client application:

- The Java API classes located in the `rt.jar` file in the `jre/lib` directory of your JDK installation.
- The Orbix Java API classes located in the `OrbixWeb.jar` file in the `lib` directory of your Orbix Java installation.
- The `config` directory of your Orbix Java installation.
- Any other classes required by the application.

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 approach is recommended, and is the approach used in "Creating the Applet" on page 44. The Orbix Java configuration files are 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 the ***Orbix Administrator's Guide Java Edition*** for more details on the Orbix Java configuration files.

Security Issues for Java Applets

Java applets are subject to important security restrictions that are imposed by the Java environment and Web browsers. The severity of these restrictions is often dependent on browser technology. Refer to the ***Orbix Administrator's Guide Java Edition*** for details about using Orbix Java on the Internet.

Learning more about Orbix Java

Part II and Part III of this guide describe Orbix Java features in more detail and expand on the information presented in Part I. Specifically, Part II and Part III include the following:

- An overview of the structure of distributed applications.
- An introduction to IDL and the corresponding mapping of IDL to the Java programming language. Both client and server programmers must be familiar with this mapping.
- Further examples of using Orbix Java to define an interface to a system component and write client and server programs.
- How to make objects available in Orbix Java, using the CORBA-defined Naming Service and the Orbix Java -specific `bind()` method.
- The use of inheritance when defining IDL interfaces, allowing an interface to be defined by extending others.
- More details on compiling IDL definitions, and registering Orbix Java servers in the Implementation Repository.
- Details on enabling communication between independently developed implementations of the CORBA standard, using IIOP (Inter-ORB Interoperability Protocol).

Part IV and Part V of this guide discuss advanced features that extend the power of Orbix Java, for example:

- *Filters* can be installed in your 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. You can write *Smart proxies* 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 Orbix Java fails to find an object being sought by a client or server, it informs *loader* objects, which can load the object from some persistent store. Interfacing Orbix Java to a persistent store, therefore, involves writing a loader object and installing this within programs that directly use that persistent store. As a result, Orbix Java is not tied to using any specific persistent store from a particular vendor.
- Orbix Java 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.
- Some applications, such as browsers, must be able to use all of the interfaces defined in a system—even those interfaces that did not exist when the browser was compiled. Orbix Java supports such applications via its *Dynamic Invocation Interface*.

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

Part II

CORBA Programming with Orbix Java

In this part

This part contains the following:

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:

1. Define a Java class that can be accessed through the IDL interface.
2. Create objects of that class within an Orbix Java 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 Orbix Java 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 the information that clients need to access the object across a network.

Consider the example of an interface that 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. See the ***Orbix Administrator's Guide Java Edition*** appendices for a full 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);
        ...
    };
};
```

Orbix Java does not guarantee that a oneway operation call will succeed. Thus, if a oneway operation fails, a client may never know. There is only one circumstance in which Orbix Java indicates failure of a oneway operation. If a oneway operation call fails *before* Orbix Java transmits the call from the client address space, Orbix Java 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 10 on page 57 illustrates the inheritance hierarchy for this interface.

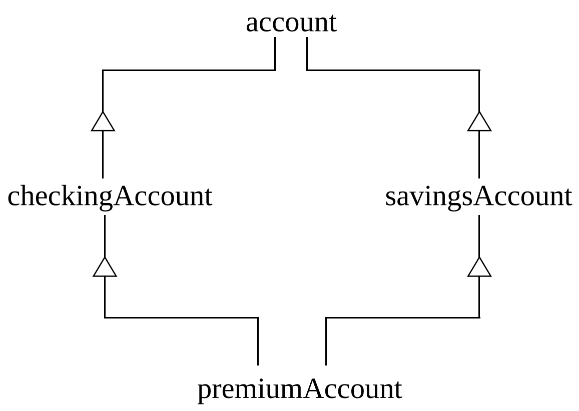


Figure 10: *Multiple Inheritance of IDL Interfaces*

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 that 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 ***Orbix Programmer's Reference Java Edition***. 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 that 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.

Table 1: *The IDL Basic Types*

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.
`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.

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 229 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, 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 65) 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.

Fixed Types

The fixed data type allows you to represent a number in two parts: a *digit* and a *scale*. The digit represents the length of the number, and the scale is a non-negative integer that represents the position of the decimal point in the number, relative to the rightmost digit.

```
module finance {
        typedef fixed<10,4> ExchangeRate;

        struct Rates {
                    ExchangeRate USRate;
                    ExchangeRate UKRate;
                    ExchangeRate IRRate;
        };
};
```

In this case, the ExchangeRate type has a digit of size 10, and a scale of 4. This means that it can represent numbers up to (+/-)999999.9999.

The maximum value for the digits is 31, and scale cannot be greater than digits. The maximum value that a fixed type can hold is equal to the maximum value of a double.

Scale can also be a negative number. This means that the decimal point is moved scale digits in a rightward direction, causing trailing zeros to be added to the value of the fixed. For example, fixed <3,-4> with a numeric value of 123 actually represents the number 1230000. This provides a mechanism for storing numbers with trailing zeros in an efficient manner.

Note: Fixed <3, -4> can also be represented as fixed <7, 0>.

Constant fixed types can also be declared in IDL. The digits and scale are automatically calculated from the constant value. For example:

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

This yields a fixed type with a digits value of 7, and a scale value of 6.

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", and to the ***Orbix Programmer's Reference Java Edition***.

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. Refer to the IDL Reference in the ***Orbix Programmer's Reference Java Edition*** 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.

IDL to Java Mapping

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

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

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 Orbix Java, the IDL specified interfaces are mapped to Java, using the Orbix Java 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.

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 that support parameter passing modes. Refer to "Mapping for Basic Data Types" on page 69.

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 70 for details.

Mapping for Interfaces and Operation Parameters

IDL *interfaces* are mapped to Java interfaces and classes that 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 72.

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 75.

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 88.

Mapping for Strings

IDL *strings*, both bounded and unbounded, map to the Java type `String`. Orbix Java performs bounds checking for `String` parameter values passed as bounded strings to IDL operations. Refer to "Mapping for Strings" on page 92.

Mapping for Sequences and Arrays

IDL *sequences,* both bounded and unbounded, map to Java arrays of the same name. Orbix Java performs bounds checking for bounded sequences. Helper and holder classes are generated for mapped IDL sequences. Refer to "Mapping for Sequences" on page 93.

IDL *arrays* map directly to Java arrays of the same name. Orbix Java performs the bounds checking, because Java arrays are not bounded. Refer to "Mapping for Arrays" on page 95.

Mapping for Fixed Types

IDL *fixed types* map to the Java `java.math.BigDecimal` class. Refer to "Mapping for Fixed Types" on page 95.

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 96.

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 97.

Mapping for Exceptions

IDL *standard system exceptions* are mapped to Java `final` classes that extend `org.omg.CORBA.SystemException` and provide access to IDL exception code. IDL *user-defined exception* types map to a `final` class that derives from `org.omg.CORBA.UserException`. User-defined exceptions have helper and holder classes generated. Refer to "Mapping for Exception Types" on page 97.

Mapping for Basic Data Types

The IDL basic data types are mapped to corresponding Java types as shown in Table 2.

Table 2: *Mapping for Basic Types*

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	

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 75.
- **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 229.

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 within 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 ***Orbix Administrator's Guide Java Edition*** 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 Orbix Java, the org.omg.CORBA set of classes represents the OMG standard abstract runtime. The actual implementation of the Orbix Java 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 Orbix Java. 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 Orbix Java 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 that 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 Orbix Java 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, because 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 Orbix Java 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() 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 bind() method is a feature specific to Orbix Java. If you wish to use only those features defined in the CORBA specification, you should compile your IDL using the -jOMG switch.)

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

```
     // in file accountHelper.java
     // Java generated by the Orbix Java IDL compiler
     //
     import org.omg.CORBA.Any;
     import org.omg.CORBA.Object;
     import org.omg.CORBA.TypeCode;
     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){
             ...
         }
         public static final account bind(String markerServer) {
             ...
         }
         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){
             ...
```

```
          }
5         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 229 for more details.
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.
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 Orbix Java.

   The Naming Service is the preferred method for locating objects in servers.

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
("accMarker:serverName", hostname);
```

This returns a proxy object that 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 84 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 Orbix Java 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, because 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 Orbix Java 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
                                ("BankMarker: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, “Marker”);
                    ...
}
```

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

Server Implementation Mapping

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

- A *Java Skeleton* class, with the name format `_<type>Skeleton.java`, used internally by Orbix Java 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`, that 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. (This is the TIE approach, which is specific to Orbix Java. If you wish to use only those features defined in the CORBA specification, you should compile the IDL using the `-jOMG` switch).
- 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 Orbix Java, and is used to support implementation using the TIE approach.

Approaches to Interface Implementation

Orbix Java 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 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 that 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:

```
// Java generated by the Orbix Java IDL compiler
//  _accountImplBase.java
//
import IE.Iona.OrbixWeb.Features.LoaderClass;
public abstract class _accountImplBase
    extends _accountSkeleton implements account {

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

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

A sample class, that 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

The IDL compiler generates a Java interface that defines the minimum set of methods that you must supply in order to implement an IDL interface using the TIE approach. The TIE approach is specific to Orbix Java. 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 Orbix Java 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 Orbix Java 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 “Using and Implementing IDL Interfaces” on page 103 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 Orbix Java 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 that 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(“Marker”);
    account x = new _tie_account
                    (accountImpl, “Marker”);
```

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
(“BankMarker:bankServer", hostname);
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), “Marker”);
        }
        ...
        m_acc = a;
        return a;
    }
}
```

If the operation newAccount() returned the account object reference as an inout or out parameter value, you must pass the generated class accountHolder to the newAccount() Java method. accountHolder is a class that 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 151.

IDL interfaces support both single and multiple inheritance. On the client side, the Orbix Java IDL compiler maps IDL interfaces to Java interfaces, which also support single and multiple inheritance, and generates Java classes that 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 Orbix Java 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 is as follows:

```
// Java generated by the Orbix Java 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 that you write that 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 87.

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 that derives from another, define an implementation class that 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 can 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, if you know that a reference of type account actually references an object that 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 Orbix Java 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 Orbix Java IDL compiler

1       public final class Fruit {
2       public static final int _apple = 0;
3       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 enums 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 enums, in this case FruitHolder.

Because 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 Orbix Java 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 Orbix Java IDL compiler
    // Clock.java
    import org.omg.CORBA.Object;
    import ClockPackage.Time;
1   import ClockPackage.TimeHolder;
2   public interface Clock extends Object {
3   public void updateTime(Time current);
4   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, you should therefore 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 Orbix Java 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`) that 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;
      ...

4     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 strings map to the Java type java.lang.String. As a Java String is fundamentally unbounded, Orbix Java 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 Orbix Java 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, Orbix Java 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 preceding example, the IDL compiler produces the following generated classes; one for the bounded sequence, and another for the unbounded sequence:

```
        // Java generated by the Orbix Java 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 performed on the mapped array during operation invocations.

Mapping for Arrays

IDL arrays map directly to Java arrays. However, Java arrays are not bounded; therefore, Orbix Java 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 Orbix Java 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 Fixed Types

The IDL fixed type maps to the Java class java.math.BigDecimal. The way IDL fixed types map to Java depends on whether or not they are declared within an IDL interface.

Fixed Types Declared outside an IDL Interface

The following sample IDL shows a fixed type declared *outside* an IDL interface:

```
// IDL
const fixed myFixed = 9999.99;
typedef fixed<6, 2> fixedIn;
```

The const myFixed is mapped to a single Java file called myFixed.java. This creates a java.math.BigDecimal called value, which is initialized to 9999.99

The typedef fixedIn is mapped to a <name>Helper file and a <name>Holder file, as is normal for other typedef types.

Fixed Types Declared within an IDL Interface

The following sample IDL shows a fixed type declared *within* an IDL interface:

```
// IDL
interface exchangeRate{
  const fixed myFixed = 9999.99;
  typedef fixed<6, 2> fixedIn;
};
```

The const myFixed is handled in a file named exchangeRate.java (the <interface name>.java file). The typedef Helper and Holder files are in a Java package directory as usual.

Refer to "Fixed Types" on page 64 for more details of this IDL type.

Mapping for Constants

The way IDL constants map to Java depends on whether or not they are declared within an IDL 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 Orbix Java 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 that 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. Because 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 Orbix Java 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 96. 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 *Orbix Programmer's Reference Java Edition*.

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 Orbix Java 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 Orbix Java 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);
        ...
}
```

Orbix Java exception handling is described in detail in “Exception Handling” on page 143.

Naming Conventions

IDL identifiers are mapped to an identifier of the same name in Java. There are, however, certain names that 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 to 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 2 on page 69 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 3 shows the mapping for the IDL parameter passing modes and return types. Refer to "Holder Classes and Parameter Passing" on page 75 for more details. All type that are not user-defined Holders are in `org.omg.CORBA`.

Table 3: *Mapping for Parameters and Return Values*

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>
string	String	StringHolder	StringHolder	String
wstring	String	WstringHolder	WstringHolder	String
sequence	array	<type>Holder	<type>Holder	array
array	array	<type>Holder	<type>Holder	array

Table 3: *Mapping for Parameters and Return Values*

IDL Type	In	Inout	Out	Return
Pseudo-IDL Types				
`NamedValue`	`NamedValue`	`NamedValueHolder`	`NamedValueHolder`	`NamedValue`
`TypeCode`	`TypeCode`	`TypeCodeHolder`	`TypeCodeHolder`	`TypeCode`
`object reference`	`<type>`	`<type>Holder`	`<type>Holder`	`<type>`

Using and Implementing IDL Interfaces

This chapter describes how servers can create objects that implement IDL interfaces, and explains how clients can access these objects through IDL interfaces. It shows how to use and implement CORBA objects through a detailed description of the banking application introduced in "Developing Applications with Orbix Java".

Overview of an Example Application

In the banking example, an Orbix Java server creates a single distributed object that represents a bank. This object manages other distributed objects that represent customer accounts at the bank.

A client contacts the server by getting a reference to the bank object. This client then calls operations on the bank object, instructing 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, where one type of distributed object acts as a factory for creating another type of distributed object, is very common in CORBA.

The source code for the example described in this chapter is available in the `demos\common\BankSimpleTie` directory of your Orbix Java installation.

Overview of the Programming Steps

The programming steps are outlined as follows:

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

Subsequent chapters add further functionality to the IDL interfaces defined in this chapter; for example, user-defined exceptions and inheritance. At this stage, the basic interfaces are sufficient to illustrate the main points.

Defining IDL Interfaces to Application Objects

This example uses two IDL interfaces: an interface for the bank object created by the server, and an interface that allows clients to access the account objects created by the bank.

The IDL interfaces are defined as follows:

```
// IDL
// In BankSimple.idl

module BankSimpleTie {

        typedef float CashAmount;
        interface Account; // forward reference
            // A factory for Bank accounts.
        interface Bank {
            // Create new account with specified name.
            Account create_account(in string name);
            // Find the specified account.
            Account find_account(in string name);
        };

        interface Account {
            readonly attribute string name;
            readonly attribute CashAmount balance;

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

In this example, the server creates a `Bank` object that accepts operation calls such as `create_account()` from clients. The operation `create_account()` instructs the `Bank` object to create a new `Account` object in the server. The operation `find_account()` instructs the `Bank` object to find an existing `Account` object.

All of the objects (both `Bank` and `Account` objects) are created in a single server process. A real system could use several different servers and many server processes.

Compiling IDL Interfaces

It is assumed that the `BankSimple.idl` source file is compiled using the following IDL compiler command:

```
idlj -jP Demos BankSimple.idl
```

See the chapter "IDL to Java Mapping" for more details on the classes generated by the IDL to Java compiler.

Implementing the IDL Interfaces

Orbix Java 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. However, both approaches can be used in the same server, if required.

This section briefly describes how you can implement an interface using both of these approaches. Refer to "Comparison of the ImplBase and TIE Approaches" on page 124 for more details.

Note: The choice of implementation method in an Orbix Java server does not affect the coding of client applications.

The TIE Approach to Implementing Interfaces

The TIE approach to defining an implementation class is shown in Figure 11 on page 105.

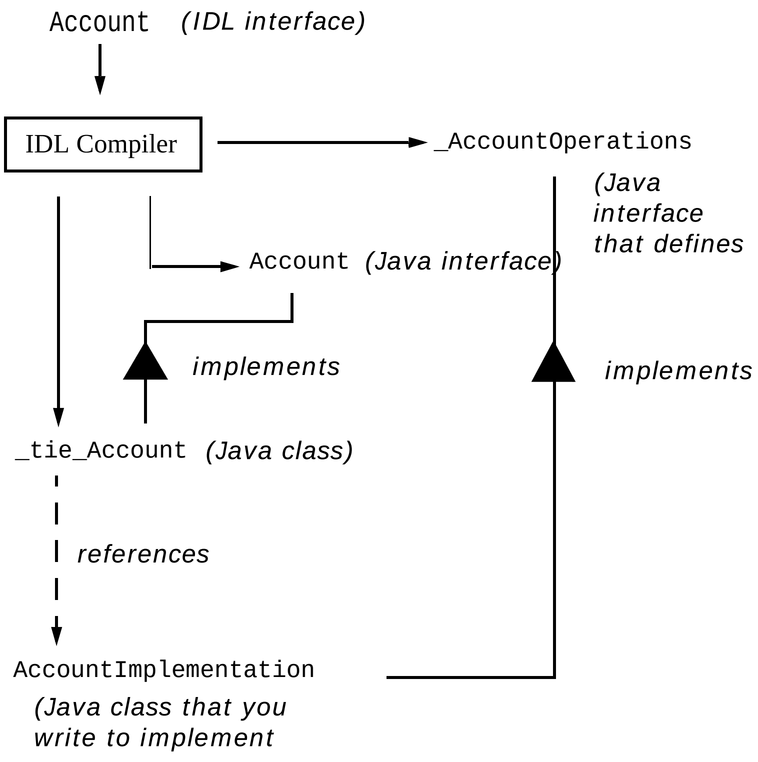


Figure 11: *The TIE Approach to Defining an Implementation Class*

Using the TIE approach, you can implement the IDL operations and attributes in a class that 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_` prefixed 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 that implements the IDL interface is passed as a parameter to the TIE class constructor.

To use the TIE approach you must 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`.

Instantiating TIE Objects

To instantiate an object of type `_tie_Account`, pass an object of type `AccountImplementation` to the TIE class constructor; in this case, `_tie_Account()`.

A TIE object is thus created that delegates incoming operation invocations to the methods of your `AccountImplementation` object.

Interface `_AccountOperations` generated by the IDL compiler is as follows:

```
// Java generated by the Orbix Java IDL compiler.

package Demos.BankSimpleTie;

public interface _AccountOperations  {
    public String name();
    public float balance();
    public void deposit(float amount) ;
    public void withdraw(float amount) ;
}
```

The ImplBase Approach to Implementing Interfaces

For each IDL interface, Orbix Java 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 Orbix Java IDL compiler produces the Java interface `Account` and the Java class `_AccountImplBase`.

Figure 12 shows the ImplBase approach to implementing IDL interfaces for the Account interface.

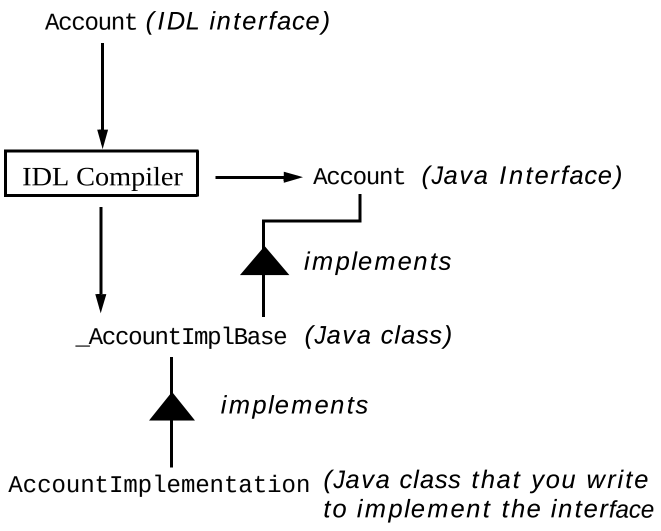

Account (IDL interface)
IDL Compiler
Account (Java Interface)
implements
_AccountImplBase (Java class)
implements
AccountImplementation (Java class that you write to implement the interface

Figure 12: *The ImplBase Approach to Defining an Implementation Class*

This chapter gives an overview of the ImplBase approach. Throughout the rest of this guide, the TIE approach to implementing IDL interfaces is used. The TIE approach is the method of choice for the majority of Java applications.

Developing the Server Application

In this section, the banking example is used to illustrate both the TIE and ImplBase approaches. The error handling necessary for a full banking application has been omitted; for example, checking if the account is overdrawn. Refer to "Exception Handling" on page 143 for details.

The following Java classes are used to implement the `Bank` and `Account` IDL interfaces:

`AccountImplementation`	Implements the `Account` IDL interface.
`BankImplementation`	Implements the `Bank` IDL interface.

Implementing the Bank Interface

This section implements the Bank IDL interface using both the TIE and ImplBase approaches.

Using the TIE Approach

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.

You must notify Orbix Java that this class implements the IDL interface by creating an object of the TIE class, which is also generated by the IDL compiler.

Using the TIE approach, you can write the code for the Bank implementation class as follows:

```
// Java
// In file BankImplementation.java.
package Demos.BankSimpleTie;

import IE.Iona.OrbixWeb._OrbixWeb;
import org.omg.CORBA.ORB;
import org.omg.CORBA.SystemException;
import java.util.*;

public class BankImplementation
    implements _BankOperations {
    // Constructor for Bank implementation object.
    public BankImplementation (org.omg.CORBA.ORB Orb) {
        m_orb = Orb;
        m_list = new Hashtable();
    }

    // Implementation for IDL operation create_account()
    public Account create_account (String name) {
        Account m_account = null;
        AccountImplementation m_account_impl = null;

        if ( m_list.get ( name ) != null ) {
        System.out.println ( "- Account for " + name + "
                already exists, " + "finding details." );
            return find_account ( name );
     }

        System.out.println ( "- Creating new account
             for "+ name + "." );

        // Create a new account.
        try {
            m_account_impl =  new AccountImplementation
                                             (name, 0.0F);
            m_account = new _tie_Account
                             (m_account_impl, “Marker”);
            m_orb.connect ( m_account );
        }

        catch ( SystemException se ) {
            System.out.println ( "[ Exception raised
                when creating Account. ]" );
        }

        // Add account to table
        m_list.put ( name, m_account );
        return m_account;
    }

    // Implementation for IDL operation find_account().
    public Account find_account (String name) {
        Account m_acc = null;
        m_acc = (Account) m_list.get (name);

        if ( m_acc == null ) {
            // account not in table.
            System.out.println ("- Unable to find
                    Account for " + name + ".");
        }
        return m_acc;
    }

    // Reference to the ORB.
    private org.omg.CORBA.ORB m_orb = null;

    // Table of accounts.
    private Hashtable m_list;
}
```

The BankImplementation class implements the _BankOperations Java interface generated by the IDL compiler.

The IDL-defined method create_account() creates an AccountImplementation object and then passes this object to the TIE class constructor, _tie_Account(). The create_account() method returns an object that implements Java interface Account. This IDL-generated type defines the client view of the IDL interface Account.

Using the ImplBase Approach

Using this approach, you must indicate that a Java class implements a specific IDL interface by inheriting from the corresponding ImplBase class generated by the IDL compiler. You can write the ImplBase code for the Bank implementation class as follows:

```
// Java
// In file BankImplementation.java.

package Demos.BankSimpleImplBase;

import IE.Iona.OrbixWeb._OrbixWeb;
...

public class BankImplementation
    extends _BankImplBase {
    // Constructor for Bank implementation object.
    public BankImplementation (ORB Orb) {
        // Same as for the TIE approach.
    }

    // Implementation for IDL operation create_account().
    public Account create_account (String name){
        Account m_account = null;
        ...
        // Create a new account
        try {
            m_account =  new AccountImplementation
                                        (name, 0.0F);
            m_orb.connect ( m_account );
        }
        catch ( SystemException se ) {
            System.out.println ( "[ Exception raised when
                                 creating Account. ]" );
        }
    ...
}
```

The BankImplementation class inherits the _BankImplBase Java class generated by the IDL compiler.

The IDL-defined method create_account() creates an AccountImplementation object and returns an object that implements Java interface Account.

Implementing the Account Interface

This section implements the Account IDL interface using both TIE and ImplBase examples.

Using the TIE Approach

When using the TIE approach, your account class implementation must implement the _AccountOperations interface generated by the IDL compiler.

The AccountImplementation class is coded as follows:

```
// Java
// In file AccountImplementation.java.
package Demos.BankSimpleTie;

public class AccountImplementation
    implements _AccountOperations {

    public AccountImplementation(String name,float bal){
        this.m_name = name;
        m_balance=bal;
        System.out.println ("- Creating account for " +
            m_name + ". Initial " + "balance of £" + bal );
    }

    // Implementation for IDL name attribute.
    public String name(){
        return m_name;
    }

    // Implementation for IDL balance attribute.
    public float balance() {
        return m_balance;
  }

    // Implementation for IDL operation deposit().
    public void deposit (float amount) {
        System.out.println ( "- Depositing £" + amount
             + " into " + m_name + "'s account" );
            m_balance += amount;
        }

        // implementation for IDL operation withdraw().
        public void withdraw(float amount) {
            System.out.println("- Withdrawing £" + amount
                 + " from " + m_name + "'s account" );
            m_balance -= amount;
        }

        // Account user's name.
        private String m_name = null;

        // Account user's balance
        private float m_balance = 0.0F;
    }
```

Using the ImplBase Approach

When using the TIE approach, your account class implementation must inherit the `_AccountImplBase` class generated by the IDL compiler. The `AccountImplementation` class is coded as follows:

```
    // Java
    // In file AccountImplementation.java.

    package Demos.BankSimpleImplBase;

    public class AccountImplementation
        extends _AccountImplBase {
        ...
    }
```

This class is identical, in every other respect, to the `AccountImplementation` class used for the TIE approach.

Writing the Server

This section shows the code for the banking server, using both TIE and ImplBase examples.

Using the TIE Approach

To create a bank implementation object, the server must pass the constructor for the bank implementation class to the TIE constructor, `_tie_Bank()`. You can implement the server using the TIE approach as follows:

```
// Java
// In file Server.java

package Demos.BankSimpleTie;

// Import Naming Service wrapper methods.
import Demos.IT_DemoLib.*;
import IE.Iona.OrbixWeb.Features.Config;
import IE.Iona.OrbixWeb._OrbixWeb;
import org.omg.CORBA.*;

public class Server {
    public static void main ( String args[] ) {
        // Initalize the ORB.
        org.omg.CORBA.ORB Orb = ORB.init (args, null);
        // Create a new bank Server
        new Server ( Orb );
    }

    // Server constructor.
    public Server ( org.omg.CORBA.ORB Orb ) {
        m_orb = Orb;
        ...
        // Create a new Naming Service wrapper.
        try {
            m_ns_wrapper = new IT_NS_Wrapper ( m_orb,
                                  m_demo_context_name );
            m_ns_wrapper.initialise();
        }
        catch ( org.omg.CORBA.UserException userEx ) {
            ...
        }
        String serverName = new String ( "IT_Demo
                                    /BankSimple" );

        // Create a new server implementation object.
        m_bank = new _tie_Bank
                      (new BankImplementation(m_orb));
        try {
         m_ns_wrapper.registerObject ( "Bank", m_bank );
        }
        catch ( org.omg.CORBA.UserException userEx ) {
            ...
        }

        // Wait for client connections.
        try {
            _OrbixWeb.ORB ( m_orb ).processEvents
                                    (10000 * 60 );
        }
        catch ( SystemException se ) {
            ...
        }
    }
...
}
```

Using the ImplBase Approach

Using the ImplBase approach, the server must create a new bank implementation object by passing a reference to the server ORB to the constructor for the `BankImplementation` class:

```
// Java
// In file Server.java

package Demos.BankSimpleImplBase;

public class Server {
    ...
        // Create a new server implementation object.
        m_bank = new BankImplementation ( m_orb );
    ...
}
```

This class is identical, in every other respect, to the `Server` class used for the TIE approach.

Object Initialization and Connection

An implementation object must be connected to the Orbix Java runtime before it can handle incoming operation invocations.

There are two ways to connect implementation objects to the Orbix Java runtimes:

- Using `ORB.connect()` and `ORB.disconnect()`.

  These methods are the CORBA-defined way of connecting an implementation to the runtime.
- Using `BOA.impl_is_ready()`.

  This is an Orbix Java -specific way of connecting implementation objects to the runtime.

Using 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 Orbix Java 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 *Orbix Programmer's Reference Java Edition* for more details.

As an example, consider the following code, that 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));

    orb.connect(mybank);
    ...
    orb.disconnect(mybank);
```

Note: ORB.connect() is automatically called when you instantiate an Orbix Java object. However, for strict CORBA compliance, you should explicitly call ORB.connect() in your application code.

Using BOA.impl_is_ready()

A server is normally coded so that it initializes itself and creates an initial set of objects. It then calls impl_is_ready() to indicate that it has completed its initialization and is ready to receive operation requests on its objects. 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 impl_is_ready() method 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);
```

The Server Name Parameter

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 Orbix Java daemon process, the server name is already known to Orbix Java 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 Orbix Java. The normal way to do this is using the first parameter to impl_is_ready(). To allow a server to be launched either automatically or manually, you should specify the serverName parameter.

By default, Orbix Java servers must be registered in the Implementation Repository, using the putitj command. Therefore, if an unknown server name is passed to impl_is_ready(), the call is rejected. However, the Orbix Java daemon can be configured to allow unregistered servers to be run manually. Refer to "Registration and Activation of Servers" on page 189 for more details on the Orbix Java daemon and the putitj command.

The Timeout Parameter

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 Orbix Java 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 use 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 ***Orbix Programmer's Reference Java Edition*** 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.

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 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 Orbix Java.

The ORB.connect() / ORB.disconnect() approach complies with the CORBA specification defined in the OMG IDL to Java mapping. Using this approach, Orbix Java 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 Orbix Java 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 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 to process 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 "Loaders" on page 319 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 the ***Orbix Administrator's Guide Java Edition*** for details on in-process servers. By default, servers are activated out-of-process.

You can combine the two approaches used for connecting to the ORB. In fact, if you call BOA event-processing operations, a combined approach is used. `ORB.connect()`is implicitly called when the implementation object is instantiated. Also, in Orbix Java, several threads can concurrently call `processEvents()`.

Note: Disconnecting the last object by default causes all BOA event-processing calls to exit.

Developing the Client Application

From the point of view of the client, the functionality provided by the banking application 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 are important concepts in distributed systems.

This section discusses developing the client application in terms of these three 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 Orbix Java occur when normal Java method calls are made on proxies.

Obtaining a Reference to a Bank Object

The banking client uses Naming Service wrapper methods to find and obtain a reference to a Bank object. Remote function invocations can then be made on the object. These concepts are illustrated in the following code extracts from the client application:

```
// Java
// In file Client.java

package Demos.BankSimpleTie;

import Demos.IT_DemoLib.*;
import Demos.BankInterface.BankGUIFrame;
import IE.Iona.OrbixWeb._OrbixWeb;
import IE.Iona.OrbixWeb._CORBA;
import org.omg.CORBA.ORB;
...

public class Client {
    public static void main ( String args[] ) {

        // Initilize the ORB
        org.omg.CORBA.ORB Orb = ORB.init ( args,null );
        // Create a new client
        new Client ( Orb );
    }

    // Client constructor.
    public Client (org.omg.CORBA.ORB Orb){
        super ( Orb, m_account_types );
        m_orb = Orb;
        m_client_frame = new ClientGUIFrame(this, m_orb);
    }
    // Connects to the bank
    public void connectToBank() {
        // Get the host name from the user interface.
        String host = m_client_frame.Get_HostName();
        m_client_frame.printToMessageWindow
                                ("Hostname got "+host);

    // Set the naming service host name.
    _OrbixWeb.ORB ( m_orb ).setConfigItem(
                                "IT_NAMES_SERVER_HOST", host );

    // Create a new naming service wrapper.
    try {
        m_ns_wrapper = new IT_NS_Wrapper ( m_orb,
                                m_demo_context_name );
    }
    catch ( org.omg.CORBA.UserException userEx ) {
        m_client_frame.printToMessageWindow ( "[ Exception
        raised during creation of naming" +
        "service wrapper.]" );
    }
    try {
        org.omg.CORBA.Object obj =
                        m_ns_wrapper.resolveName ("Bank") ;
        m_bank = BankHelper.narrow (obj);
        m_client_frame.printToMessageWindow("Connection
                                           succeeded." );
    }
    catch ( org.omg.CORBA.UserException userEx ) {
        m_client_frame.printToMessageWindow ( "[ Exception
        raised getting Bank reference " + userEx + "]" );
    }
    ...
}
```

Alternatives to the Naming Service

Using the Naming Service is the CORBA-defined way to establish communications with a particular object. There are two other ways that a client can obtain a reference to an object that it needs to communicate with:

- Using a return value or an out parameter to an IDL operation call.
- Using the Orbix Java -specific bind() mechanism.

Using a Return Value or an Out Parameter

A client can also receive an object reference as a return value or as an out parameter to an IDL operation call. This results in the creation of a proxy in the client's address space. Operation create_account(), for example, returns a reference to an Account object, and a client that calls this operation can then make operation calls on the new object.

Using the Orbix Java specific bind Method

The following code sample shows how a client could obtain a reference to a Bank object using the Orbix Java specific bind() operation:

```
// 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
("BankMarker:Bank", hostname);
}
catch (org.omg.CORBA.SystemException ex) {
    System.out.println
    ("Exception during bind : " + ex.toString());
}
System.out.println
    ("Connection to " + hostname + " succeeded.\n");
```

The bind mechanism is implemented by the static member method bind() of the BankHelper class generated by the IDL compiler. This method takes a parameter that specifies the location of the required implementation object in the system. Orbix Java can choose any Bank object within the named server.

The value returned by BankHelper.bind() is a proxy object reference.

Making Remote Invocations

The proxy object reference returned by the Naming Service 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 generated code for interface Account is as follows:

```
// Java generated by the IDL compiler

package Demos.BankSimpleTie;

public interface Account
    extends org.omg.CORBA.Object {

    public String name();

    public float balance();

    public void deposit(float amount) ;

    public void withdraw(float amount) ;

    public java.lang.Object _deref() ;
}
```

The generated code for interface Bank is as follows:

```
// Java generated by the IDL compiler

package Demos.BankSimpleTie;
public interface Bank
    extends org.omg.CORBA.Object {

    public Demos.BankSimpleTie Account
        create_account(String name) ;

    public Demos.BankSimpleTie Account
        find_account(String name) ;

    public java.lang.Object _deref() ;
}
```

Both Java types inherit from the Java interface org.omg.CORBA.Object. This is an Orbix Java interface that defines functionality common to all IDL object reference types. Refer to the API Reference in the ***Orbix Programmer's Reference Java Edition*** on org.omg.CORBA.Object for further information on this extra functionality.

Registration and Activation

The last step in developing and installing the banking application is to register the Bank server in the Implementation Repository.

Running the Orbix Daemon

Before registering the server, you should ensure that an Orbix C++ or Java daemon process (orbixd or orbixdj) is running on the server machine.

To run the Orbix Java daemon, enter the orbixdj command from the bin directory of your Orbix Java installation. To run the Orbix C++ daemon, enter the orbixd command.

On Windows, you can also start a daemon process by clicking on the appropriate menu item from the Orbix Java menu.

The Implementation Repository

The Orbix Java Implementation Repository records the server name and the details of the Java class that 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 that the class expects.

Every node in a network that 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 putitj command, which takes the following simplified form:

putitj *putitj switches* -java *server name*
    -classpath *classpath class name*
    command-line arguments for server

For example, you could register the Bank server as follows:

```
putitj -java Bank Demos.BankSimpleTie.Server
```

The class Demos.BankSimpleTie.Server is then registered as the implementation code for the server Bank at the current host.

The putitj command does not cause the specified server class to be interpreted. The Java interpreter can be explicitly invoked on the class, or the Orbix Java daemon can cause the class to be interpreted in response to an incoming operation invocation. It uses the Orbix Java configurable IT_DEFAULT_CLASSPATH as its classpath when searching for the class. You can specify an alternative classpath using the putitj utility. Refer to the ***Orbix Administrator's Guide Java Edition*** for more details.

Execution Trace

This section examines the events that occur when the Bank server and client are run. The TIE approach is used to show the initial trace, and the ImplBase approach is then discussed. 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 Orbix Java 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 Server.java

    public Server (org.omg.CORBA.ORB Orb) {
        m_orb = Orb;
        ...
        // Create a new server implementation object.
        m_bank = new _tie_Bank
                (new BankImplementation(m_orb),
“Marker”);
        ...
    }
```

Client Side

The client first obtains a reference to the Bank object, using the Naming Service, for example:

```
// Java
// In file Client.java.

public class Client {
    ...
    public void connectToBank() {
        ...
        org.omg.CORBA.Object obj =
                        m_ns_wrapper.resolveName ("Bank")
;

        m_bank = BankHelper.narrow (obj);
        ...
    }
}
```

When the object reference has been obtained, the Orbix Java daemon launches an appropriate process by invoking the Java interpreter on the Server class, if the process is not already running.

This results in the automatic generation of a proxy object in the client. This acts as a stand-in for the remote BankImplementation object in the server. The object reference m_Bank within the client is now a remote object reference as shown in Figure 13 on page 123.

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:

```
// Java
// In file Client.java.
// In class Client

    Account new_account = null;
    String current_name = m_client_frame.Get_UserName();

        try {
            new_account = m_bank.create_account
                                            (current_name );
        }
        catch ( SystemException se ) {
            ...
        };
```

When the m_bank.create_account() call is made, the method BankImplementation.create_account() 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, create_account() 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 new_account variable as shown in Figure 13 on page 123.

If the ImplBase approach is used, the final diagram is as shown in Figure 14 on page 123.

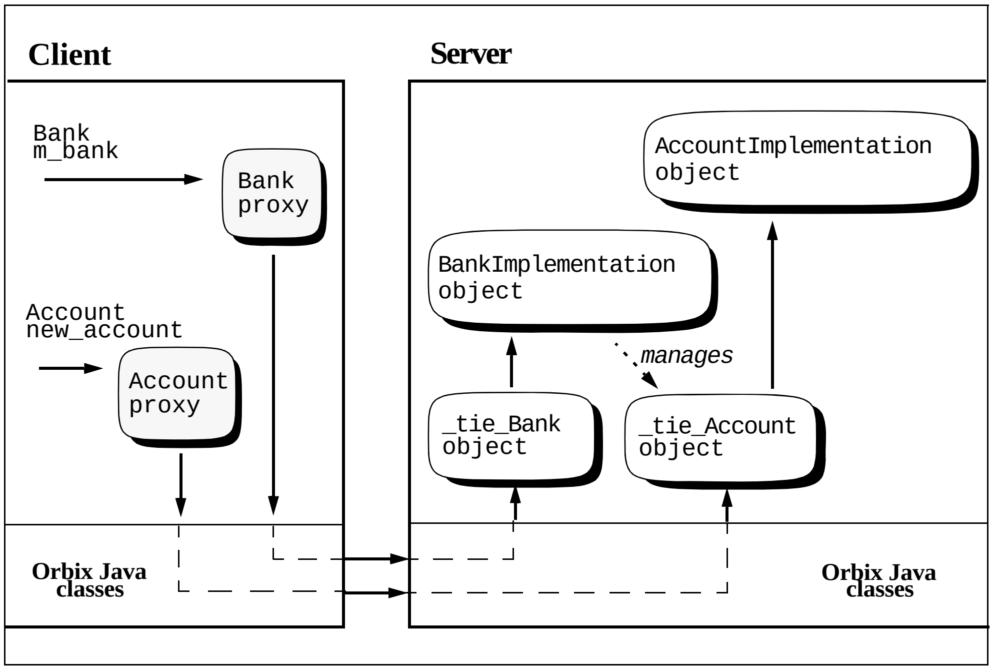


Figure 13: *Client Creates Object (TIE Object)*

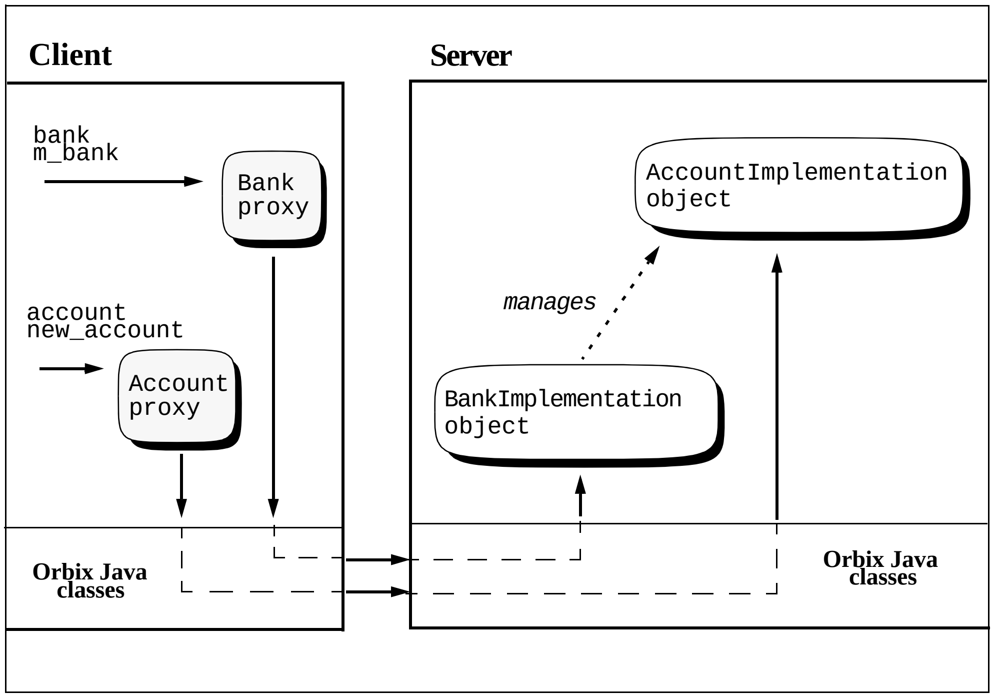


Figure 14: *Client Creates Object (ImplBase Approach)*

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 that may affect your 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 that 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 for Orbix Java applications.

The creation of a TIE object for each implementation object can 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; because both can be used within the same server.

The next two sections examine two 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 you to provide a number of different implementation 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 know which implementation class is used.

Providing Different Interfaces to the Same Implementation

Using the TIE approach, you can have a Java implementation class that implements more than one IDL interface. This class must implement the generated Java `Operations` interfaces for all the IDL interfaces it supports. The class 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, because each interface requires the implementation class to extend an IDL-generated base class.

Making Objects Available in Orbix Java

A central requirement in a distributed object system is that clients must be able to locate the objects they wish to use. This chapter describes how you can make objects available in servers and enable clients to locate these objects in clients.

Before using a CORBA object, a client must establish contact with it. To do this, the client must get an *object reference* for the required object. An object reference is a unique value that tells an ORB where an object is and how to communicate with it.

An important issue for every CORBA application is how servers can make object references available to clients, and how clients can retrieve these references to establish contact with objects. This chapter describes three solutions to this issue:

- Using the CORBA Naming Service.
- Using the Orbix Java -specific `bind()` method.
- Using object reference strings to create proxy objects.

These solutions are presented after a brief introduction to how object references work in CORBA.

Identifying CORBA Objects

Every CORBA object is identified by an object reference, which is a unique value that includes all the information an ORB requires to locate and communicate with the object. When a client gets an object reference, the ORB creates a proxy in the client's address space. When the client calls an operation on the proxy, the ORB transmits the request to the target object.

Orbix supports two protocols for communications between clients and servers:

- The CORBA standard Internet Inter-ORB Protocol (IIOP). This is the default protocol.
- The Orbix protocol.

Each of these communication protocols has its own object reference format. The Orbix protocol requires an Orbix Java 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.

Interoperable Object References

An object that is accessible via IIOP is identified by an interoperable object reference (IOR). Because 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.

Orbix Java Object References

Every object created in an Orbix Java application has an associated Orbix Java object reference. This object reference includes the following information:

- An object name that is unique within its server. This is referred to as the object's *marker*.
- The object's server name
  This is sometimes called an *implementation name* in CORBA terminology.
- The server's hostname.

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 Orbix Java object reference is fully specified by the following fields:

- Object marker.
- Server name.
- Server hostname.
- IDL interface type of the object.
- Interface Repository (IFR) server in which the definition of this interface is stored.
- IFR server host.

Accessing Object References

All Orbix Java 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();
```

Orbix Java automatically assigns the server host, server name and IDL interface fields when an object is created. It is not generally necessary to update these values.

Orbix Java also assigns a marker value to each object, but you may choose alternative marker values in order to explicitly name Orbix Java 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 Orbix Java Objects

An Orbix Java marker value allows a name (in string format) to be associated with an object, as part of its object reference. There are two ways to assign markers to Orbix Java objects:

- Assigning a marker on creation of the object.
- Renaming an object using `_marker()`.

Assigning a Marker on Creation

You can specify a marker name at the time an object is created. If you do not specify a marker for a newly created object, a name is automatically chosen by Orbix Java. 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 an 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) {
            ...
        }
    }
```

Renaming the Object using _marker()

You can use the modifier method `_marker(String)` to rename an object which has a user-specified name or a name assigned by Orbix Java. This is defined in the interface `ObjectRef` in package `IE.Iona.OrbixWeb.CORBA`. For details on how to convert an Orbix Java object to an instance of `ObjectRef`, refer to the class `_OrbixWeb.ObjectRef` in the ***Orbix Programmer's Reference Java Edition*** .

Accessing an Object’s Marker Name

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) {
    ...
}
```

Marker Chosen by Orbix Java

The marker names chosen by Orbix Java consist of a string composed entirely of decimal digits. To ensure that your markers are different from those chosen by Orbix Java, do not use strings consisting entirely of digits.

Note: Marker names cannot contain any ‘:’ or null characters.

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, Orbix Java 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.

Using 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 object's interface, server or hostname.

This section outlines the features of OrbixNames, the Orbix full implementation of the CORBA Naming Service. The following topics are outlined:

- The interface to the Naming Service.
- Format of names within the Naming Service.
- Making initial contact with the Naming Service.
- Associating names with objects.
- Using names to find objects.
- Associating a compound name with an object.

For a complete description of using OrbixNames, refer to the ***OrbixNames Programmers and Administrator's Guide***.

The Interface to the Naming Service

The programming interface to the Naming Service is defined in IDL. A standard set of IDL interfaces allow you to access all the Naming Service features. OrbixNames, for example, is a normal Orbix Java server that contains objects that implement these interfaces.

The Naming Service interfaces are defined in 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 a system are organized into a naming graph, that 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.

Compound Names

A compound name in the Naming Service takes the more abstract form of an IDL sequence of name components. In addition, the name components that 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 OrbixNames.

The type `Istring` is a placeholder for a future IDL internationalized string that 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 permitted.

Making Contact with the Naming Service

The IDL interface `NamingContext`, defined in module `CosNaming`, provides access to most features of the Naming Service. The first step in using the Naming Service is to get a reference to an object of this type.

Each Naming Service contains a special `CosNaming::NamingContext` object called the root naming context. This acts as an entry point to the service. The root naming context allows you to create new naming contexts, bind names to objects, resolve object names, and browse existing names.

An application can obtain a reference to its root 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().

Associating Names with Objects

Once you have a reference to the root naming context, you can begin to associate names with objects. The operation CosNaming::NamingContext::bind() enables you to bind a name to an object in your application. This operation is defined as:

```
void bind (in Name n, in Object o)
    raises (NotFound, CannotProceed,
            InvalidName, AlreadyBound);
```

To use this operation, you first create a CosNaming::Name structure containing the name you want to bind to your object. You then pass this structure and the corresponding object reference as parameters to bind().

Using Names to Find Objects

Given an abstract name for an object, you can retrieve a reference to the object by calling CosNaming::NamingContext::resolve(). This operation is defined as:

```
Object resolve (in Name n)
    raises (NotFound, CannotProceed, InvalidName);
```

When you call resolve(), the Naming Service retrieves the object reference associated with the specified CosNaming::Name value and returns it to your application.

The return type of the resolve() operation is an 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.

Associating a Compound Name with an Object

If you want to use compound names for your objects, you must first create naming contexts. For example, consider the compound name shown in Figure 15.

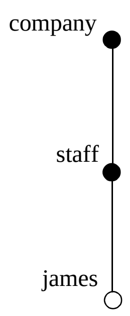


Figure 15: *An Example Compound Name*

To create this compound name:

1. Create a naming context and bind a name with identifier `company` (and no kind value) to it.
2. Create another naming context, in the scope of the `company` context, and bind the name `staff` to it.
3. Bind the name `james` to your application object in the scope of the `staff` context.

The operation `CosNaming::NamingContext::bind_new_context()` enables you to create naming contexts:

```
NamingContext bind_new_context (in Name n)
    raises (NotFound, CannotProceed,
            InvalidName, AlreadyBound);
```

To create a new naming context and bind a name to it, create a `CosNaming::Name` structure for the context name and pass it to `bind_new_context()`. If the call is successful, the operation returns a reference to your newly created naming context.

You should refer to the ***OrbixNames Programmers and Administrator's Guide*** for detailed Java examples of using the Naming Service.

Federation of Name Spaces

The collection of all valid names recognized 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 16 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.

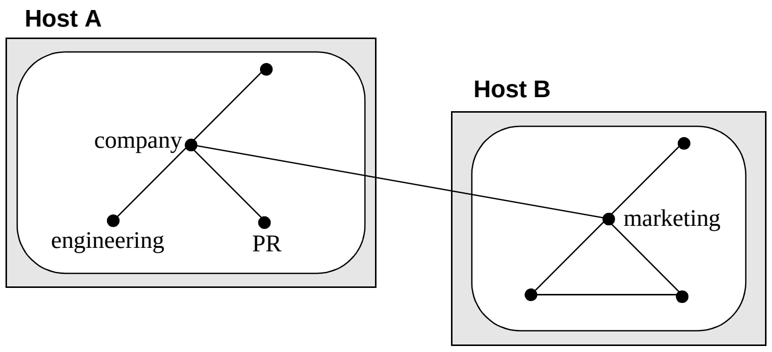


Figure 16: *Naming Graph Spanning Different Name Servers*

Binding to Objects in Orbix Java Servers

Note: This section discusses the use of the Orbix Java -specific `bind()` method to create proxy objects in clients. This should not be confused with the CORBA-specified `bind()` method for use with the Naming Service. Orbix Java Edition only supports fully qualified Orbix Java -specific bind. That is bind (“`myMarker:myServer`”, `hostname`).

There is a difference between binding to Orbix Java 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 Orbix Java `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 Orbix Java server. The `bind()` method cannot be used with objects that 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, Orbix Java 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, Orbix Java is allowed some freedom when choosing the object.

The bind() Method

The bind() method is a static method automatically generated by the IDL compiler for each IDL Java class. 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:

```
// In file BankHelper.java
// Java generated by the Orbix Java IDL compiler.

package Demos.BankSimple;

import IE.Iona.OrbixWeb._OrbixWeb;
...

public class BankHelper {
        ...

        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) {
            ...
        }
    }
}
```

Parameters to bind()

The bind() method is overloaded and takes the following sets of parameters:

- markerServer, host
- markerServer, host, orb
- A full object reference as returned by the method org.omg.CORBA.ORB.object_to_string().

The orb parameter to bind() 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.

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 Implementation Repository. It is not necessarily the name of a class or an interface although you can assign a server the same name as that of a class or interface. The Implementation Repository is described in detail in "Registration and Activation of Servers" on page 189.

Orbix Java 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 ':'.

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 marker must be supplied in all cases. Anonymous bind (i.e. not supplying a marker) is deprecated in Orbix Java Edition. However, clients built with previous versions of OrbixWeb can still use anonymous bind even with Orbix Java Edition servers.

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 141 for more details.

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.

Example Calls to bind()

This section shows some sample calls to `bind()`.

1. Bind to the `College_Green` object at the AIB server at node `beta`, in the internet domain `mc.ie`. The object should implement the Bank IDL interface.

   ```
   Bank b = BankHelper.bind
       ("College_Green:AIB", "beta.mc.ie")
   ```

2. Bind to the `College_Green` object at the AIB server at Internet address `10.59.0.1`. The object should implement the Bank IDL interface.

   ```
   Bank b = BankHelper.bind
       ("College_Green:AIB", "10.59.0.1");
   ```

Binding and Exceptions

By default, `bind()` raises an exception if the desired object is unknown to Orbix Java. This requires Orbix Java to ping the desired object in order to check its availability The ping operation is defined by Orbix Java and has no effect on the target object. The pinging causes the target Orbix Java server process to be activated if necessary, and confirms that this server recognizes the target object.

If you wish to improve efficiency by reducing the number of remote invocations, ping can be disabled by calling the method `pingDuringBind()` as follows:

```
// Java
import IE.Iona.OrbixWeb._CORBA;
...
_CORBA.Orbix.pingDuringBind(false);
```

When ping 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 ping is enabled.

Using Object Reference Strings to Create Proxy Objects

An Orbix Java object is uniquely identified by an object reference. Given a stringified form of an Orbix Java object reference, an Orbix Java 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 that 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 reference 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 ***Orbix Programmer's Reference Java Edition*** 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 Orbix Java Dynamic Invocation Interface (DII).

Exception Handling

The implementation of an IDL operation or attribute can throw an exception to indicate that a processing error has occurred. This chapter describes Orbix Java exception handling in detail, using a banking example. This example builds on the concepts illustrated in the banking example in the chapters "Developing Applications with Orbix Java", and "Using and Implementing IDL Interfaces".

There are two types of exceptions that an IDL operation can throw:

- *User-defined exceptions*.

  These exceptions are defined explicitly in your IDL definitions, and can only be thrown by operations.
- *System exceptions*.

  These are pre-defined exceptions that all operations and attributes can throw.

This chapter describes user-defined exceptions and system exceptions, and shows how to throw and catch these exceptions.

Orbix Java 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. It also describes the Orbix Java mapping for such user-defined exceptions. The source code for the example described in this chapter is available in the `demos\common\BankExceptions` directory of your Orbix Java installation.

The IDL Definitions

In this example, the `create_account()` operation can raise an exception if the bank cannot create an `Account` object. The exception `CannotCreate` is defined within the `Bank` IDL interface. This defines a string member that indicates the reason why the `Bank` rejected the request:

```
        // IDL
        // In file bankexceptions.idl

        module BankExceptions {
            typedef float CashAmount;
            interface Account;

            interface Bank {
1           // User-defined exceptions.
                exception CannotCreate { string reason; };
                exception NoSuchAccount { string name; };

                Account create_account (in string name)
2                       raises (CannotCreate);
                Account find_account (in string name)
                    raises (NoSuchAccount);
            };

            interface Account {
                // User-defined exception.
3                   exception InsufficientFunds { };
                    readonly attribute string name;
                readonly attribute CashAmount balance;
                void deposit (in CashAmount amount);
                void withdraw (in CashAmount amount)
                    raises (InsufficientFunds);
            };
        };
```

This IDL is explained as follows:

1. `CannotCreate` and `NoSuchAccount` are user-defined exceptions defined for the `Bank` IDL interface.
2. Operation `BankExceptions::Bank::create_account()` can raise the `BankExceptions::Bank::CannotCreate` exception. It can only raise listed user-defined exceptions. It can raise any system-defined exception.
3. An exception does not need to have any data members.

Note: Read or write access to any IDL attribute can also raise any system-defined exception.

The Generated Java Code

This chapter assumes that the IDL source file is compiled using the following command:

```
idlj -jP Demos BankExceptions.idl
```

The IDL compiler generates Java code within the Demos.BankExceptions package. For example, the following Java class is generated for the IDL definition for the CannotCreate exception:

```
    // Java generated by the Orbix Java IDL compiler.

    package Demos.BankExceptions.BankPackage;

1   public final class CannotCreate
        extends org.omg.CORBA.UserException
        implements java.lang.Cloneable {

        public String reason;

        public CannotCreate() {
            super();
        }
2   public CannotCreate(String reason) {
            super();
            this.reason = reason;
        }
        ...
    }
```

1. The class CannotCreate inherits from org.omg.CORBA.UserException. This Orbix Java class in turn inherits from java.lang.Exception. This inheritance allows CannotCreate to be thrown and handled as a Java exception, using try...catch blocks.
2. Because the CannotCreate exception has one member (reason, of type String) the generated class provides a constructor that initializes this member.

The generated Java interface for Bank is as follows:

```
// Java generated by the Orbix Java IDL compiler.

package Demos.BankExceptions;

public interface Bank
    extends org.omg.CORBA.Object {

    public Account create_account(String name)
        throws CannotCreate;

    public Account find_account(String name)
        throws NoSuchAccount;
    ...
}
```

System Exceptions

The CORBA specification defines a set of system exceptions to which Orbix Java adds a number of additional exceptions. These system exceptions can be raised during Orbix Java 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 Orbix Java 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 150.

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) {
        ...
    }
```

Refer to the ***Orbix Administrator's Guide Java Edition*** for a list of system exceptions defined by Orbix Java.

Obtaining Information from 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 Orbix Java class `CompletionStatus` (in the package `org.omg.CORBA`).

The return 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.

Example of Server-Side Exception Handling

All Orbix Java exceptions inherit from Java class java.lang.Exception. Consequently, the rules for throwing Orbix Java exceptions follow those for throwing standard Java exceptions: you must throw an object of the exception class. For example, you can use the following code to throw an exception of IDL type Bank::CannotCreate:

```
    // Java
    import Demos.BankExceptions.BankPackage;
        ...
        throw new CannotCreate("Some reason");
```

This uses the automatically generated constructor of class CannotCreate to initialize the exception object's reason member with the string "Some reason".

The implementation of the create_account() operation in class BankImplementation can be coded as follows:

```
// Java
// In file BankImplementation.java,

package Demos.BankExceptions;

import Demos.BankExceptions.BankPackage.*;
import org.omg.CORBA.SystemException;
...

// Implemetation for the Bank IDL interface.
public class BankImplementation
    implements _BankOperations {
    ...
    // Implementation for IDL operation create_account().
    public Account create_account(String name)
        throws CannotCreate {

        Account account_ref = null;

        // Raise an exception if account already exists.
```

```
            if (m_list.get(name) != null) {
                System.out.println("- Account for " + name + "
                        already exists, " + "throwing CannotCreate
                                            exception." );
                throw new CannotCreate("Account for " + name + "
                                           already exists.");
            }

            // Raise an exception if bank is full.
            if (m_account_count >= MAX_ACCOUNTS) {
                throw new CannotCreate("No more space for new
                                                accounts ");
            }

            System.out.println("- Creating new account for " +
                                                 name + ".");

            // Create a new account
            try {
                account_ref =
                    new _tie_Account(new AccountImplementation
                        (name,0F, m_currency_format),name);
                m_orb.connect(account_ref);
            }
            catch(SystemException se){
                System.out.println("[ Exception raised when
                    creating Account. " + se + " ]");
            }
        ...
        }
    ...
    }
```

Example of Client-Side Exception Handling

A client calling an operation that raises 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 code extract shows client-side exception handling with the CannotCreate user-defined exception:

```
// Java
// In file Client.java

package Demos.BankExceptions;

import Demos.BankExceptions.Bank;
import Demos.BankExceptions.BankPackage.*;
...

public class Client {
    ...
    public void createAccount(String name) {
        if (!"".equals(name)) {
            if (m_bank_ref != null) {
                try {
                   m_bank_ref.create_account(name);
                   m_client_frame.printToMessageWindow
                    ("Created account for "+ name +"." );
                }
                catch(CannotCreate cc) {
                 m_client_frame.printToMessageWindow ("[
                    Cannot create account " + cc.reason
                                        + " ]");
                }
                catch(SystemException se) {
                m_client_frame.printToMessageWindow("[
                   Cannot create account " + se +" ]");
            }
        }
        ...
    }
}
```

Handling Specific System Exceptions

A client can also provide a handler for a specific system exception. For example, to explicitly handle a COMM_FAILURE exception, you could write the following code:

```
// Java

import org.omg.CORBA.SystemException;
import org.omg.CORBA.COMM_FAILURE;
....

public class Client {
    ...
        try {
                org.omg.CORBA.Object obj =
                m_ns_wrapper.resolveName(“Bank”);
                m_bank_ref = BankHelper.narrow(obj);
                m_client_frame.printToMessageWindow
                                    ("Connection succeeded.");
        }
        catch (COMM_FAILURE cfe) {
            m_client_frame.printToMessageWindow
                ("Unexpected communication failure
                                        exception:" + cfe);
        }
        catch (SystemException se) {
            m_client_frame.printToMessageWindow
                ("Unexpected system exception" + se );
        }
    ...
    }
}
```

This code is described as follows:

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 because of implicit casting.
3. If you only wish to know the type of exception that occurred, the message output from class SystemException is sufficient. A handler for an individual exception is required only when specific action is to be taken if that exception occurs.

Using Inheritance of IDL Interfaces

This chapter describes how to implement inheritance of IDL interfaces, using a banking example. This example builds on the concepts illustrated in the banking examples in the chapters "Using and Implementing IDL Interfaces" and "Exception Handling".

You can define a new IDL interface that uses functionality provided by an existing interface. The new interface inherits or derives from the base interface. IDL also supports multiple inheritance, allowing an interface to have several immediate base interfaces. This chapter shows how to use inheritance in Orbix Java using the banking example.

The source code for the example described in this chapter is available in the `demos\BankInherit` directory of your Orbix Java installation.

Single Inheritance of IDL Interfaces

The IDL for this example demonstrates the use of single inheritance of IDL interfaces. It expands the banking example in "Exception Handling" on page 143 to enable support for checking (current) accounts.

The IDL Interfaces

The IDL interfaces to the banking example are now defined as follows:

```
   // IDL
   // In file bankinherit.idl

   #include "bankexceptions.idl"

   module BankInherit {
       interface CheckingAccount; // forward reference

       // BetterBank manufactures checking accounts.
1      interface BetterBank : BankExceptions::Bank {
           // New operation to create checking accounts.
2          CheckingAccount create_checking (in string name,
           in BankExceptions::CashAmount overdraft)
               raises(CannotCreate);
       };

       // New CheckingAccount interface.
3      interface CheckingAccount : BankExceptions::Account {
           readonly attributeBankExceptions::CashAmount
           overdraft;
       };
   };
```

This IDL can be explained as follows:

1. `BetterBank` inherits the operations of `BankExceptions::Bank` and adds a new operation to create checking accounts. You do not need to list the account operations from `BankExceptions::Bank` because these are now inherited.
2. The new `create_checking()`operation added to interface `BetterBank` manufactures `CheckingAccounts`.
3. The new interface `CheckingAccount` derived from interface `BankExceptions::Account`. `CheckingAccount` has an overdraft limit, and the implementation allows the balance to become negative.

The Client-Side Generated Types

It is assumed that the IDL definition is compiled using the following command:

```
idlj -jP Demos bankinherit.idl
```

Orbix Java maps IDL interfaces to Java interfaces. The IDL interface inheritance hierarchy maps directly to the Java interface inheritance hierarchy, as shown in Figure 17:

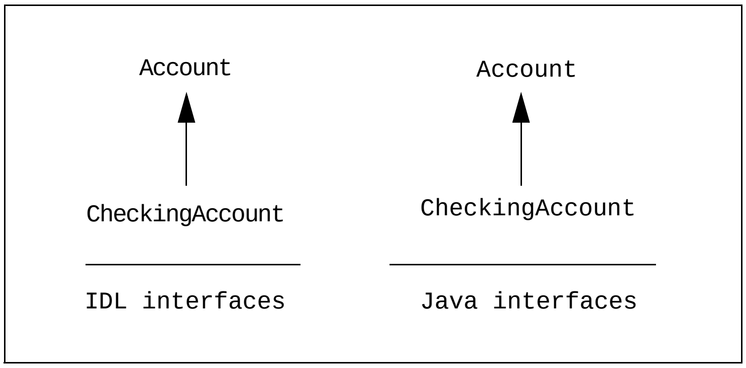


Figure 17: *IDL and Java Inheritance Hierarchies*

IDL-Generated Java Interfaces

The IDL interface `Account` maps to the following Java interface:

```
// Java
// Automatically generated
// in file Account.java

package Demos.BankExceptions;

public interface Account
    extends org.omg.CORBA.Object {

    public String name();
    public float balance();
    public void deposit(float amount);
    public void withdraw(float amount)
        throws InsufficientFunds;
    ...
}
```

The IDL interface `CheckingAccount` maps to the following Java interface:

```
// Java
// Automatically generated
// In file CheckingAccount.java

package Demos.BankInherit;

public interface CheckingAccount
    extends BankExceptions.Account {

    public float overdraft();
    ...
}
```

As with the IDL interface `CheckingAccount`, the mapped Java interface `CheckingAccount` inherits the methods contained in interface `Account`.

IDL-Generated Java Classes

The IDL compiler also generates Java implementation classes for the Java interfaces. These Java implementation classes provide client proxy functionality for the IDL operations. This proxy functionality facilitates the distribution of objects in Orbix Java. In addition, the IDL compiler also generates a Java *helper* class that implements the static `bind()` and `narrow()` methods. Refer to “IDL to Java Mapping” on page 67 for a full description of the mapped Java classes.

IDL interface inheritance maps directly to the inheritance hierarchy of the generated Java interfaces, but it does not map to the generated Java classes for those interfaces. Therefore, each Java class that 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 Orbix Java 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 159.

The generated Java class that implements the Account interface is as follows:

```
// Java
// In file _AccountStub.java

package Demos.BankExceptions;

public class _AccountStub
    extends org.omg.CORBA.portable.ObjectImpl
    implements Account {

    public _AccountStub () {}

    public String name() {
        ...
    }

    public float balance() {
        ...
    }

    public void deposit(float amount) {
        ...
    }

    public void withdraw(float amount)
         throws InsufficientFunds {
        ...
    }
    ...
}
```

The generated Java class that implements the CheckingAccount interface is as follows:

```
// Java
// In file _CheckingAccountStub.java

package Demos.BankInherit;

public class _CheckingAccountStub
    extends org.omg.CORBA.portable.ObjectImpl
    implements CheckingAccount {

    public _CheckingAccountStub () {}

    public float overdraft() {
        ...
    }

    public String name() {
        ...
    }

    public float balance() {
        ...
    }

    public void deposit(float amount) {
        ...
    }
```

```
        public void withdraw(float amount)
            throws InsufficientFunds {
            ...
        }
        ...
    }
```

The _AccountStub and _CheckingAccountStub classes enable client method calls to be forwarded to the server.

Using Inheritance in a Client

You can create and manipulate instances of CheckingAccount in a similar way to the instances of Account in "Developing the Client Application" on page 116. For example, the following code extract shows how to create CheckingAccount objects:

```
    // Java
    // In file Client.java

    package Demos.BankInherit;

    public class Client {
        ...
        public void createCheckingAccount(String name,
                                        float overdraftAmount
    ) {
            try {
                m_BankRef.create_checking(name,

    overdraftAmount);
                m_clientFrameReference.printToMessageWindow(
                    "Created checking account for " +name+
    "." );
            }
            catch (CannotCreate ex){
                ...
            }
            catch (SystemException se) {
                ...
            }
        }
        ...
    }
```

The IDL-defined create_checking() method creates a CheckingAccount object with the specified name and overdraft.

Using Inheritance in a Server

This section uses a banking 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 124 for a detailed discussion of both approaches.

The TIE Approach

Using the TIE approach to implementing IDL interfaces, the `CheckingAccount` implementation class simply implements Java interface `_CheckingAccountOperations`.

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 implements any of the required methods.

On the server side, the IDL compiler generates the Java interface `_CheckingAccountOperations`. This defines the methods that a server class must implement in order to support IDL interface `CheckingAccount`. This Java interface inherits from type `_AccountOperations`, which serves a similar purpose for IDL type `Account`.

Because the inherited class `AccountImplementation` implements the methods defined in interface `_AccountOperations`, you need only implement methods that differ in class `CheckingAccImplementation`; you can reuse common functionality.

For example, class CheckingAccImplementation calls the constructor for AccountImplementation. The IDL-defined name() and balance() accessor methods are not re-implemented:

```
// Java
// In file CheckingAccImplementation.java
package Demos.BankInherit;

import Demos.BankExceptions._AccountOperations;
import Demos.BankExceptions.AccountPackage.
    InsufficientFunds;

public class CheckingAccImplementation
    extends AccountImplementation
    implements _CheckingAccountOperations {

    // Constructor.
    public CheckingAccImplementation(String name,
        float bal, float overdraft) {
        // Calls AccountImplementation constructor.
        super (name, bal);

        m_Overdraft = overdraft;
        m_OverdraftLimit = overdraft;
    }


    //Implementation for deposit() now updates overdraft.
    public void deposit(float amount) {
        ...
    }
    // Implementation for withdraw() updates overdraft.
    public void withdraw(float amount)
        throws Demos.BankExceptions.AccountPackage.
        InsufficientFunds {
        ...
    }

    // Implementation for new IDL operation.
    public float overdraft() {
        return m_Overdraft;
    }
}
```

Because class CheckingAccImplementation inherits from class AccountImplementation, all Account methods do not need to be re-implemented. Using the TIE approach enables you to take advantage of the reuse characteristics of object-oriented programming.

The ImplBase Approach

The IDL compiler generates the abstract class _CheckingAccountImplBase. This supports the ImplBase approach to IDL interface implementation. To implement IDL interface CheckingAccount using the ImplBase approach, define a Java class that inherits from class _CheckingAccountImplBase, 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 CheckingAccount (for example, an existing class also implements IDL type Account), this class cannot be reused in the CheckingAccount implementation class.

The CheckingAccount implementation class *must* directly implement all the operations of IDL interface CheckingAccount and all interfaces from which it inherits. This restriction severely limits the flexibility of the ImplBase approach.

Using the ImplBase approach, class CheckingAccImplementation cannot inherit the Account implementation, so you must re-implement the existing Account methods before adding any new functionality.

```
// Java
package Demos.BankInherit;

public class CheckingAccountImplementation
    extends _CheckingAccountImplBase {
    // Re-implement existing Account methods and
    // add new CheckingAccount methods.
}
```

Refer to "Using and Implementing IDL Interfaces" on page 103 for details of implementing using the ImplBase approach.

Multiple Inheritance of IDL Interfaces

IDL supports multiple inheritance of interfaces. The following serves as an example:

```
// IDL
interface Account {
    readonly attribute string name;
    readonly attribute CashAmount balance;

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

// Derived from interface Account.
interface CheckingAccount : Account {
    readonly attribute float overdraft;
};

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

// Indirectly derived from interface Account.
interface PremiumAccount :
    CheckingAccount, 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 that define client-side functionality. For example, the interface hierarchy in the preceding definition maps as shown in Figure 18.

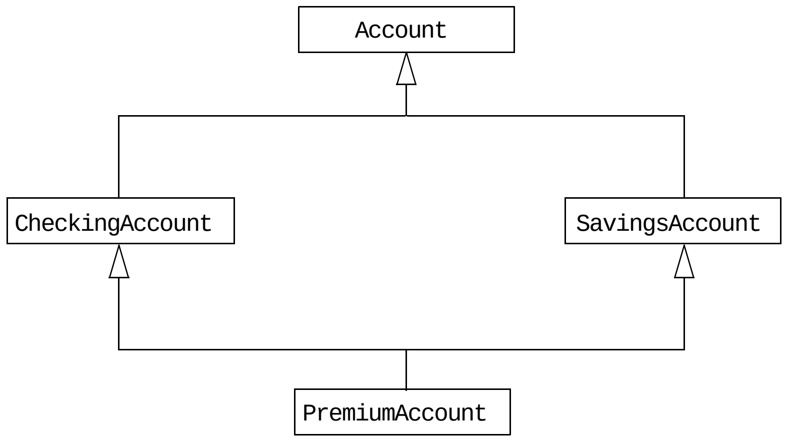


Figure 18: *Multiple Inheritance of IDL Interfaces*

The inheritance hierarchy does not map to the Java classes that 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`, `CheckingAccount`, and `DepositAccount`) directly on that reference.

Implementing Multiple Inheritance

On the server side, the implementation class requirements are identical to those for single inheritance. You can implement multiple inheritance in your sever using either the TIE approach or the ImplBase approach.

Using The TIE Approach

Using the TIE approach, the implementation class must implement Java interface `_PremiumAccountOperations`, but can also 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 cannot 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.

Using The ImplBase Approach

Using the ImplBase approach, when implementing type `PremiumAccount` you must inherit from class `_PremiumAccountImplBase` and directly implement all methods for interface `PremiumAccount` and all types from which it inherits.

Callbacks from Servers to Clients

Orbix Java clients usually invoke operations on objects in Orbix Java servers. However, Orbix Java 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 Orbix Java. This chapter describes a common approach to implementing callbacks in an Orbix Java 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 Orbix Java

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 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" 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 Orbix Java 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
         (opsMarker:opsServer", hostname);

         // 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 19.

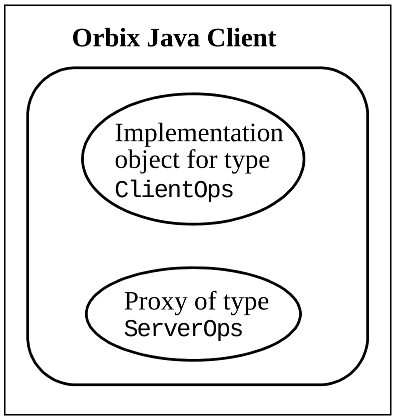


Figure 19: *Client Objects*

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 20.

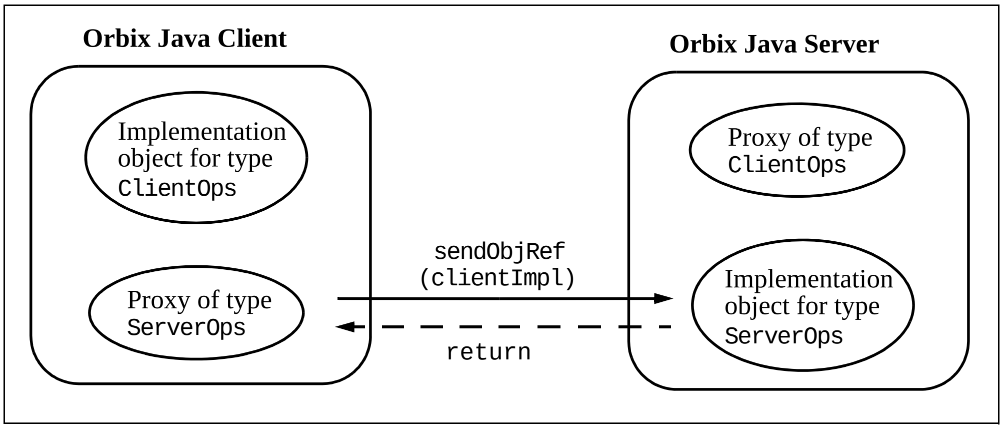


Figure 20: *Client Passes Implementation Object Reference to Server*

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 ***Orbix Programmer's Reference Java Edition*** 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.

Writing a Server

You can code the server application as a normal Orbix Java server. Specifically, you should define an implementation class for type `ServerOps`, and create one or more implementation objects.

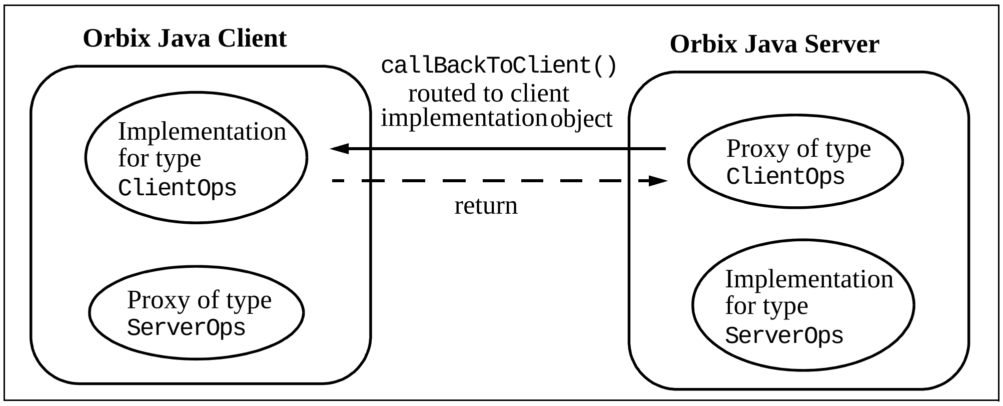
The implementation of the method `sendObjRef()` for type `ServerOps` requires special attention. This method receives an object reference from the client. When this object reference enters the server address space, a proxy for the client's `ClientOps` object is created. The server will use this proxy 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 21.



Orbix Java Client
Orbix Java Server
callBackToClient() routed to client implementation object
Implementation for type ClientOps
Proxy of type ClientOps
return
Proxy of type ServerOps
Implementation for type ServerOps

Figure 21: *Server Invokes Operation on Client's Callback Object*

The transmission of requests from server to client is possible because Orbix Java 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 Orbix Java daemon. Therefore, the client can process the callback event without being registered in the 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.

Orbix Java 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 Orbix Java 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 Orbix Java 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 Orbix Java 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 22.

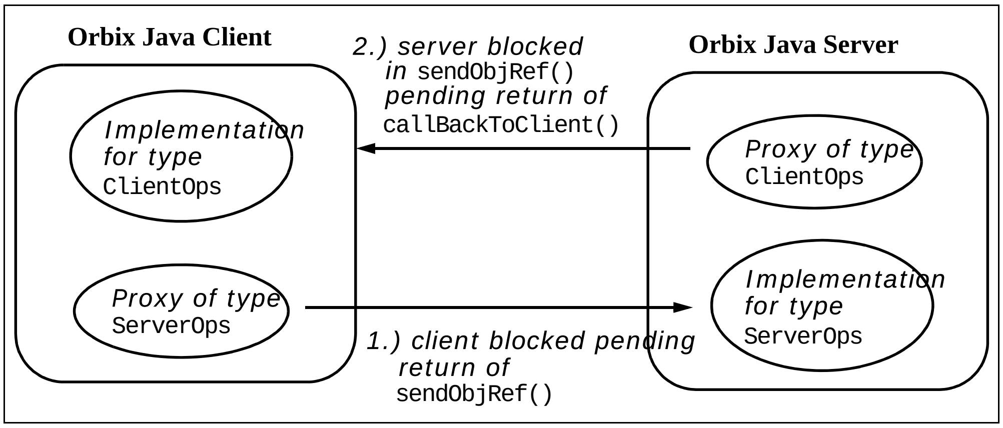

Orbix Java Client
Orbix Java Server
2.) server blocked in sendObjRef() pending return of callBackToClient()
Implementation for type ClientOps
Proxy of type ClientOps
Proxy of type ServerOps
Implementation for type ServerOps
1.) client blocked pending return of sendObjRef()

Figure 22: *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 Orbix Java 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 Orbix Java 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 Orbix Java 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 22, 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 Orbix Java 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 Orbix Java DII deferred synchronous approach to invoking operations. As described in the chapter "Dynamic Invocation Interface", 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" 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 Orbix Java application may create multiple threads of execution. To avoid deadlock, it may be useful to create a separate thread dedicated to handling Orbix Java events.

For example, an Orbix Java 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 Orbix Java 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

Orbix Java 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.

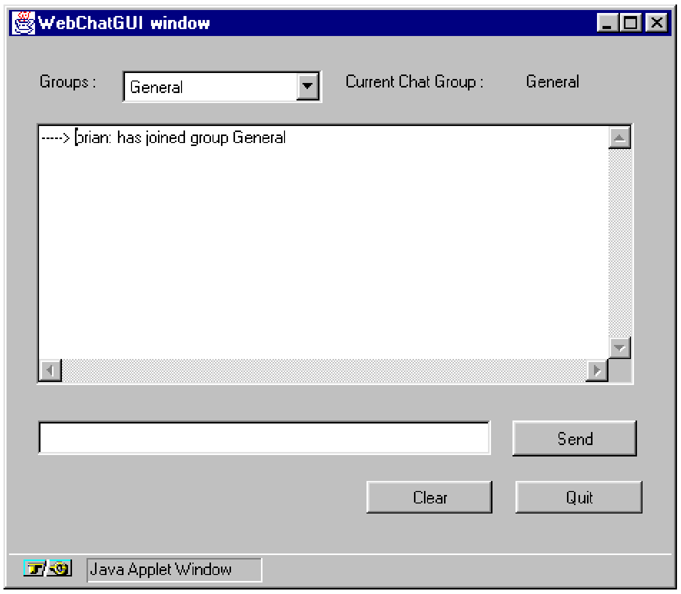
These event processing methods are defined on Orbix Java class `BOA` (in package `IE.Iona.OrbixWeb.CORBA`). If the client is to receive callbacks, the client’s `ORB` object must be initialized as type `BOA`. The client call, for example, to, `processEvents()` blocks while waiting for incoming Orbix Java 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/orbixjava/WebChat` directory of your Orbix Java installation.

Users join a chat group by downloading an Orbix Java 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 23 on page 169.



-----> brian: has joined group General

Figure 23: *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 is provided—the **WebChat Administrator Server**. This displays statistical information about the messages in the system. This 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 Orbix Java 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);
     }
  }
}
```

Implementing the 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 Orbix Java 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 Orbix Java 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 Orbix Java 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 Orbix Java events are processed, including server callback invocations. This class is defined as follows:

```
// Java
// In package WebChat,
// in class WebChatGUI.

// Orbix Java 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".

The static `main()` method begins by retrieving command-line arguments and then instantiates an object of type `WebChatGUI`.

Implementing the 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 23 on page 169 shows the Web Chat client user interface.

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("");
}
```

The **Clear** button implementation sets both the message and main chat group text boxes to null.

```
public void clickedClearButton() {
    SendEdit.setText("");
    ChatEdit.setText("");
}
```

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);
    }
}
```

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 memember
        ...
        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;
        }
    }
}
```

A `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 Orbix Java 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 III

Running Orbix Java Programs

In this part

This part contains the following:

Running Orbix Java Clients

This chapter deals with running Orbix Java client applications and applets, and provides information on some general runtime issues for clients.

Running Client Applications

The procedure for running an Orbix Java client application is similar to the procedure for running any standalone 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 that contains the `main()` method for the application.

The only runtime difference between an Orbix Java application and a standard Java application lies in the first of these requirements. An Orbix Java 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 located in the `OrbixWeb.jar` file in the `lib` directory of your Orbix Java installation. 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 Orbix Java packages.
- The Java API classes.
- The IDL compiler output classes.
- The application-specific classes.

For example, if you are using the `java` interpreter from Oracle's JDK, you should add the location of each to the `CLASSPATH` environment variable or specify this information in the `-classpath` switch.

You must also ensure that the path to your Orbix Java `config` directory is included on the list of directories specified after the `-classpath` switch.

Orbix Java 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.pl` and `owjavac.pl`, are described in the section "Using the Orbix Java Wrapper Utilities" on page 186.

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 Orbix Java Client Applets

The requirements for running an Orbix Java 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 Orbix Java 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 Orbix Java 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 Orbix Java 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 Orbix Java 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 Orbix Java 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
    ARCHIVE=OrbixWeb.jar>
    ...
</APPLET>
```

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 Orbix Java 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 that 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 Orbix Java client applets downloaded from Web servers. In particular, such clients can only communicate directly with Orbix Java servers located on the host from which the clients themselves were downloaded. If this restriction applies to an Orbix Java 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`.

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 Orbix Java Clients

An Orbix Java client application or applet has the same fundamental characteristics as any other Java program. You can debug Orbix Java clients with any available Java debugging tool, for example, the JDK `jdb` debugger.

When debugging Orbix Java clients, it is especially important to be aware of Java exceptions thrown during Orbix Java method invocations. Orbix Java 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...catch` statements. Similarly, they can be handled like standard Java exceptions when using a Java debugger.

For more details on Orbix Java integration with Java exceptions, refer to "Exception Handling" on page 143.

Possible Platform Dependencies in Orbix Java Clients

In general, Orbix Java clients are only dependent on the availability of a Java interpreter on the target execution platform. However, you should also be aware that using the `bind()` method can affect the platform-independence of an Orbix Java system.

Using bind()

If a client uses the Orbix Java `bind()` method to create a proxy for a server object, the `bind()` call fails unless an Orbix Java 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 Orbix Java daemon is available.

Using the Orbix Java Wrapper Utilities

The Orbix Java Wrapper Utilities, `owjava.pl` and `owjavac.pl`, 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:\Micro Focus\Orbix
3.3\demos\classes;c:\Micro Focus\Orbix
3.3\lib\OrbixWeb.jar;c:\Micro Focus\Orbix
3.3\config;c:\JDK\jre\lib\rt.jar myPackage.myClass
```

Using the `owjava.pl` wrapper utility, you can reduce the standard command-line entry to the following:

```
perl owjava.pl myPackage.myClass
```

The `owjava.pl` and `owjavac.pl` wrappers use Perl scripts. Orbix Java ships with Perl provided in the `contrib` directory of your installation on 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.pl` wrapper is a front end for the Java interpreter you are using, designed for use with Orbix Java. It takes all the same arguments as your chosen Java interpreter and passes them on, together with some other defaults.

`owjava.pl` uses the `ORBIX_HOME` environment variable to find the Orbix Java configuration files. 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.pl` passes the default classpath and a variable containing the path of the configuration files to the Java interpreter. So, for example, if Orbix Java is installed in `C:\Orbix 3.3` and the JDK is installed in `C:\JDK`, calling `owjava.pl` as follows:

```
perl owjava.pl myPackage.myClass
```

executes the following command:

```
> c:\JDK\bin\java -classpath c:\Orbix 3.3\demos\classes;
c:\Orbix 3.3\lib\OrbixWeb.jar;c:\Orbix 3.3\config;
c:\JDK\jre\lib\rt.jar myPackage.myClass
```

You can override this standard behaviour by using the Orbix Java Configuration Explorer to change the settings. Refer to the ***Orbix Administrator's Guide Java Edition*** for details of the Configuration Explorer.

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 Orbix Java. Its behaviour is similar to the `owjava.pl` tool described previously, but the defaults are different. By default, `owjavac.pl` passes the default `CLASSPATH` and the classes directories to the compiler.

So, for example, if Orbix Java s installed in `c:\Orbix 3.3` and the JDK is installed in `c:\JDK`, calling `owjavac.pl` as follows:

```
perl owjavac.pl
-d c:\Orbix 3.3\demos\classes\myClass.java
```

executes the following command:

```
>c:\JDK\bin\javac -classpath c:\Orbix 3.3\demos\classes;
c:\Orbix 3.3\lib\OrbixWeb.jar;c:\Orbix 3.3\config;
c:\JDK\jre\lib\rt.jar
-d c:\Orbix 3.3\demos\classes \myClass.java
```

You can override this standard behaviour by using the Orbix Java Configuration Explorer to change the settings. Refer to the ***Orbix Administrator's Guide Java Edition*** for details of the Configuration Explorer.

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 javac Command

```
> c:\JDK\bin\javac -classpath c:\Orbix 3.3\demos\classes;
   c:\Orbix 3.3\lib\OrbixWeb.jar;c:\Orbix 3.3\config;
   c:\JDK\lib\classes.zip
   -d c:\Orbix 3.3\demos\classes myClass.java
```

Using the java Command

```
> c:\JDK\bin\java -classpath c:\Orbix 3.3\demos\classes;
c:\Orbix 3.3\lib\OrbixWeb.jar;c:\Orbix 3.3\config;
c:\JDK\jre\lib\rt.jar
myPackage.myClass
```

In addition the following ORB properties should also be passed to the `java` command.

```
-Dorg.omg.CORBA.ORBClass=IE.Iona.OrbixWeb.CORBA.ORB
-Dorg.omg.CORBA.ORBSingletonClass=IE.Iona.OrbixWeb.CORBA.
   singletonORB
```

For detailed information on the full range of Orbix Java utilities, refer to the ***Orbix Administrator's Guide Java Edition***.

Registration and Activation of Servers

This chapter describes the Implementation Repository. This is the component of Orbix Java that maintains registration information about servers and controls their activation. The Implementation Repository is effectively a database of server activation information, implemented in the Orbix Java daemon. The Orbix Java 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 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 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 Orbix Java 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 Orbix Java 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 ***Orbix Java Edition Administrator's Guide***.

When a client first communicates with an object, Orbix Java 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

Orbix Java 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 Orbix Java 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 ***Orbix Administrator's Guide Java Edition*** 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, Orbix Java routes the invocation to that process. Otherwise, Orbix Java 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 for each 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 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 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 users will cause a different process to be created for each 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 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.

Because Orbix Java 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 ***Orbix Programmer's Reference Java Edition*** 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 196 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 Orbix Java does not launch the server when an operation invocation arrives for one of its objects. This is explained in "Unregistered Servers" on page 197.

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 Orbix Java 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.

Putitj

The `putitj` 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 `putitj` command creates or modifies an activation order within the existing entry. In the latter case, the `putitj` command must specify the *same* fundamental activation mode (shared, unshared or per-method) as that already registered for the server.

Catitj

The `catitj` command displays the information on a server in an Implementation Repository entry. Alternatively, you can use the Server Manager tool. Refer to the ***Orbix Administrator's Guide Java Edition*** for details of how to use this tool.

The Orbix Java Putitj Utility for Server Registration

The `putitj` utility registers servers with the Implementation Repository. This section outlines some examples of common uses of `putitj`. A full description of `putitj` and its switches is given in the ***Orbix Administrator's Guide Java Edition***.

The `putitj` command is used most often in either of the following forms:

```
putitj serverName -java
    -classpath <full classPath> className

putitj 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 `common.cfg` configuration file, when the daemon attempts to launch the server.

The `-java` switch is an extension of the standard Orbix `putitj` command This indicates that the specified server should be launched by the Java interpreter. You can truncate this switch to `-j`.

By default, `putitj` 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, `putitj` registers a server in the *multiple-client* activation mode. This means that all client processes bind to the same server process. For example:

```
putitj 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 `putitj` command does not launch the server. You can do this explicitly from the shell or otherwise. Alternatively, Orbix Java 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:

```
putitj 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 Orbix Java Server Manager tool. Refer to the ***Orbix Administrator's Guide Java Edition*** for details on both of these methods.

Examples of Using Putitj

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:

```
putitj -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 putitj command to identify an object to which putitj applies:

```
putitj -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 putitj command.

Additional Registration Commands

Implementation Repository entries created by `putitj` can be managed using the following commands:

`catitj`	Outputs full details of a given Implementation Repository entry.
`chmoditj`	Allows launch and invoke rights on a server to be granted to users other than the server owner.
`chownitj`	Allows the ownership of Implementation Repository entries and directories to be changed.
`killitj`	Kills a running server process.
`lsitj`	Lists a specific entry or all entries.
`mkdiritj`	Creates a new registration directory. You can structure the Implementation Repository hierarchically like UNIX file names.
`pingitj`	Pings the Orbix Java daemon to determine whether it is alive.
`psitj`	Outputs a list of server processes known to the Orbix Java daemon.
`rmdiritj`	Removes a registration directory.
`rmitj`	Removes an Implementation Repository entry or modifies an entry.

Execute any of these commands without arguments to obtain a summary of its switches. Refer to the ***Orbix Administrator's Guide Java Edition*** for a complete description of each command.

Activation and Pattern Matching

A server programmer can choose the marker names for objects, as described in "Making Objects Available in Orbix Java" on page 127. Alternatively, they can be assigned automatically by Orbix Java.

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 that 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 Orbix Java. 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 Orbix Java to launching servers manually:

- `BOA.impl_is_ready()`
- `ORB.connect()`

BOA.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 Orbix Java such as calling an operation on a remote object 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.pl` 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 Orbix Java 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 the ***Orbix Administrator's Guide Java Edition*** 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 Orbix Java 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 Orbix Java servers that communicate over the standard Orbix communications protocol, and limits such servers to running on hosts where an Orbix or Orbix Java daemon process is available.

Activation Issues Specific to IIOP Servers

You do not need to register Orbix Java servers that 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 Orbix Java daemon.

Unregistered IIOP servers are important in a Java domain. This is because they can be completely independent of any supporting processes that may be platform-specific. In particular, any server that 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 Orbix Java 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. Orbix Java addresses this problem by allowing you to assign a well-known IIOP port number to the server.

These issues are discussed in detail in "ORB Interoperability" on page 213.

Security Issues for Orbix Java Servers

This section covers issues concerned with security for Orbix Java servers. The method for addressing security issues will depend, in some cases, on which Orbix Java 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. Orbix Java 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 `chmoditj` command to modify the two access control lists. However, only the owner of an Implementation Repository entry can call the `chmoditj` command on it. The original owner is the user who calls the `putitj` command. Subsequently, you can change the ownership using the `chownitj` command.

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 Orbix Java 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.

You should *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, does 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 interface IE.Iona.OrbixWeb.CORBA.BOA).
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`, which is one of the two daemon processes available in Orbix Java. The Java daemon, or `orbixdj`, currently implements a subset of the `IT_daemon` interface. Differences in implementation between `orbixd` and `orbixdj` are explained in the ***Orbix Administrator's Guide Java Edition***

The UNIX utilities, such as `putitj`, `catitj`, and the Orbix Java Server Manager are implemented in terms of the daemon's IDL interface.

You should refer to the ***Orbix Programmer's Reference Java Edition*** for a full description of the interface to the Implementation Repository.

Using the Server Manager

The Server Manager is a graphical user interface that provides much of the functionality of the Orbix Java utilities. The Server Manager facilitates Implementation Repository management, offering functionality similar to `putitj`, `rmitj`, `mkdiritj` and other command utilities. It also supports the activation and deactivation of servers. Refer to the ***Orbix Administrator's Guide Java Edition*** 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 the ***Orbix Administrator's Guide Java Edition*** for more details on the features supported by the Java daemon.

Using the Orbix Java Daemon

The Orbix Java daemon (`orbixdj`*) is a Java implementation of the* `IT_daemon` *interface. The Java daemon administers the Implementation Repository and is responsible for activating servers automatically.*

The Implementation Repository is an important component of CORBA. This stores information that can be used by the ORB to activate servers on demand from clients. In previous versions of Orbix Java, the executable `orbixd` was required to manage this repository and to activate servers. This version of Orbix Java 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 Orbix Java documentation. References to *daemon* apply to functionality supported by both `orbixd` and `orbixdj`.

Overview of the Java Daemon

The Java daemon is responsible for transparently activating Orbix Java servers, and reactivating 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 that 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 reactivate the server. When the Java daemon reactivates 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.
- Orbix Java 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 Orbix Java) and Orbix'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 the Java Daemon

You can launch the Java daemon from the Orbix Java menu in the Windows Start menu.

To launch the Java daemon from the command line, use the following command:

```
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.

Switch	Effect
-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 the Orbix Java Configuration Explorer GUI tool to customize the settings for the Java daemon. The following outlines the configuration settings that concern the Java daemon. It also indicates how these settings should be changed using the Orbix Java Configuration Explorer.

For more details on the Configuration Explorer, refer to the ***Orbix Administrator's Guide Java Edition***.

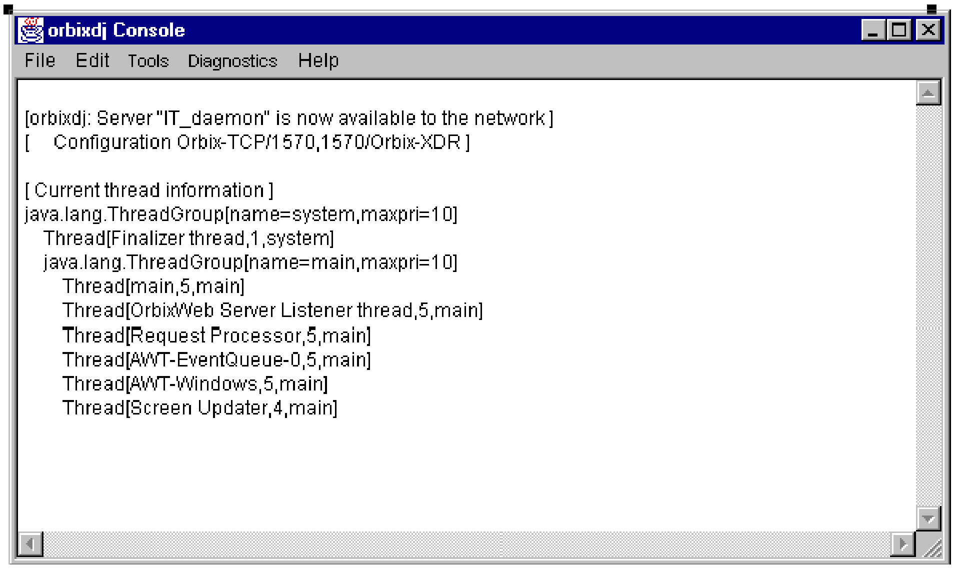
Settings	Effect
IT_IMPL_IS_READY_TIMEOUT	When an in-process server is launched, the Java daemon waits to be informed that the server is active before allowing the causative client request to proceed. Refer to "Guidelines for Developing In-Process Servers" on page 208 for further details. The Java daemon waits a maximum of this amount of time, specified in milliseconds. The default is 30,000 milliseconds, or 30 seconds.
IT_IMP_REP_PATH	This is the absolute path to the Implementation Repository.
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 that require a separate port for each protocol.
IT_DAEMON_SERVER_BASE	A server that is launched in separate processes listens on its own port. This is the value of *the first port*, and subsequently allocated ports increment. The default is 1590.
IT_DAEMON_SERVER_RANGE	Refer to IT_DAEMON_SERVER_BASE. The default is 2000.
IT_JAVA_INTERPRETER	This is the absolute path to the Java interpreter.

Settings	Effect
IT_DEFAULT_CLASSPATH	This is the classpath 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 Orbix Java classes *must* be in the CLASSPATH. There is no default.

Viewing Output with 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 24 on page 207.
Tools->Garbage Collection	Causes the Java Virtual Machine to run the garbage collector synchronously, and may free up more 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 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 level of diagnostics output to the console to REQUEST. Equivalent to calling: ORB.setDiagnostics (32).
Help->About	Displays the **About** dialog box.



```
[orbixdj: Server "IT_daemon" is now available to the network ]
[    Configuration Orbix-TCP/1570,1570/Orbix-XDR ]

[ Current thread information ]
java.lang.ThreadGroup[name=system,maxpri=10]
   Thread[Finalizer thread,1,system]
   java.lang.ThreadGroup[name=main,maxpri=10]
      Thread[main,5,main]
      Thread[OrbixWeb Server Listener thread,5,main]
      Thread[Request Processor,5,main]
      Thread[AWT-EventQueue-0,5,main]
      Thread[AWT-Windows,5,main]
      Thread[Screen Updater,4,main]
```

Figure 24: *Sample Output from* ***Tools*** **->** ***Threads*** *Menu Option*

Setting Diagnostics Levels

As with other Orbix Java 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 “Orbix Java Diagnostics” on page 223 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 scalability in terms of performance and resource consumption. These benefits include:

- 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. 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 described in the following sections:

ORB Configuration

Orbix Java 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 Orbix Java `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 ***Orbix Programmer's Reference Java Edition*** 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

Orbix Java-specific features such as filters, loaders and transformers are configured for the entire ORB. Therefore, 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. These 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 the `psit` utility.

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 the ***Orbix Programmer’s Guide Java Edition*** to see how `impl_is_ready()` can control event processing for out-of-process servers.

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 outlines those that no longer apply.

Activation

The Java daemon currently only supports shared server activation mode.

Java Version

The Java daemon requires Java version 1.1 or higher.

IT_daemon Interface

The Java daemon currently implements a large subset of Orbix's daemon IDL, `IT_daemon`. The following is a list of the methods that 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 Orbix Java Server Manager tool and the `chmodit` command-line utility to set access rights.

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.

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 specialization of GIOP that uses TCP/IP as the transport layer. Specialized 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 organization different ORBs may coexist reflecting separate development effort or different ORB requirements by different parts of the organization 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 using IIOP in a Platform-Independent Application” shows how IIOP can be used in Orbix Java.

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 defines eight message types. These formats are intended for internal use only. 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.

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 eight 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 "Interoperability between Orbix and Orbix Java" on page 221 for more details on IORs.

IIOP in Orbix Java

Orbix Java 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 Orbix Java 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:

```
idlj -m IIOPOnly
```

As described in the chapter "Making Objects Available in Orbix Java", 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. Orbix Java 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 that communicate using the Orbix protocol require an Orbix Java daemon to run on the server host. This limits these servers to platforms where an Orbix Java daemon is available. However, using IIOP, you can design client and server applications that 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 Orbix Java `bind()` method.

Refer to "Registration and Activation of Servers" for details on how Orbix Java 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 Orbix Java to create an interoperable application which does not rely on the availability of an Orbix Java 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 OrbixNames and then invokes `processEvents()` to handle client invocations on that IOR. The client retrieves the IOR using OrbixNames 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:

```
idlj -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 OrbixNames 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), “gridmarker”);
        }
        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, Orbix Java automatically identifies this object using an IOR. The server then resolves the initial context in the OrbixNames and associates the compound name `objects.math.grid` with the IOR, as described in "Making Objects Available in Orbix Java". Finally, the server enters an Orbix Java 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);
    }
}
```

Configuring an IIOP Port Number for an Orbix Java Server

Using IIOP, an Orbix Java server must listen for client connection requests on a fixed TCP/IP port. The port number for each server is assigned by Orbix Java on start-up.

In most cases this is done by the Orbix Java daemon. Refer to the descriptions of `IT_DAEMON_SERVER_BASE` and `IT_DAEMON_SERVER_RANGE` in the ***Orbix Administrator's Guide Java Edition*** 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, Orbix Java allows you to select a well-known IIOP port for each server program.

By default, the Orbix Java 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 `putitj -port` switch, for example:

```
putitj 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 Orbix Java 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 putitj command, the Orbix Java daemon assigns a default well-known port to the server.

A server which does not depend on the availability of an Orbix Java 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.

Interoperability between Orbix and Orbix Java

The default protocol for the Orbix Java 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);
```

Orbix Java Diagnostics

Orbix Java provides comprehensive diagnostics log output. This functionality is supplied by the `IE.Iona.OrbixWeb.Features.DiagnosticsLog` *API. This chapter explains how to set diagnostics levels in Orbix Java, and outlines the output from each diagnostics level.*

Setting Diagnostics

The `setDiagnostics()` method controls the level of diagnostics messages output by Orbix Java. 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 `0-255`.

The `setDiagnostics()` method returns the previous diagnostics level.

Diagnostics Levels

Orbix Java 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 Orbix Java, 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 that 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 Orbix Java to `stdout` by:

- Using the command line.
- Using the Java Daemon graphical console.
- Using the Orbix Java Configuration Explorer.

  Refer to the ***Orbix Administrator's Guide Java Edition*** for details.

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`.

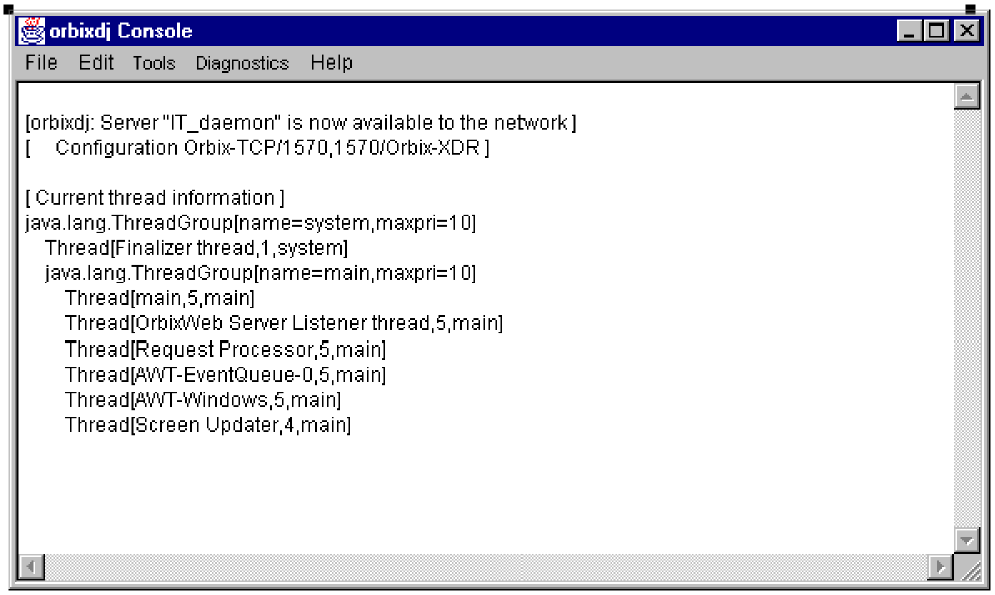
The diagnostics levels in this range are explained in "Setting Diagnostics" on page 223. Using the command line enables full diagnostics log support.

Using the Java Daemon 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. 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)`.



```
[orbixdj: Server "IT_daemon" is now available to the network ]
[    Configuration Orbix-TCP/1570,1570/Orbix-XDR ]

[ Current thread information ]
java.lang.ThreadGroup[name=system,maxpri=10]
   Thread[Finalizer thread,1,system]
   java.lang.ThreadGroup[name=main,maxpri=10]
      Thread[main,5,main]
      Thread[OrbixWeb Server Listener thread,5,main]
      Thread[Request Processor,5,main]
      Thread[AWT-EventQueue-0,5,main]
      Thread[AWT-Windows,5,main]
      Thread[Screen Updater,4,main]
```

Figure 25: *The Orbixdj Graphical Console*

Combining Diagnostics Levels

You can also use the Java Daemon graphical console to combine diagnostics levels as shown in Figure 25 on page 225.

For example, if you select the **LOW**, **ORB**, **BOA** and **Proxy** menu items, the `orbixdj` console produces a combined output for these diagnostics components.

Part IV

Advanced CORBA Programming

In this part

This part contains the following:

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 *Orbix Programmer's Reference Java Edition* 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
    ("anyMarker:anySave", hostname);

    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("anyMarker:anyServer", hostname);

    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(“anymarker:anyServer”,
   hostname);

    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(“anyMarker:anyServer”,
   hostname);

    rv = vRef.getit();

    // extract a sequence of longs
    if ((rv.type()).equal(longSeqHelper.type())) {
        toReceive = longSeqHelper.extract (rv);
    }
}
catch (SystemException se) {
    ...
}
```

Orbix Java does not destroy the value of an any after extraction. You can therefore extract the value of an any more than once.

Note: The Orbix Java -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 "Parameter Passing Modes and Return Types".

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 Orbix Java.

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 Orbix Java 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/orbixjava/grid_dsi` directory of your Orbix Java installation.

Dynamic Invocation Interface

In a normal Orbix Java 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 Orbix Java 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 Orbix Java 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.

Orbix Java supports the CORBA *Dynamic Invocation Interface* (DII). This 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 a bank example to demonstrate the use of the DII. The IDL definitions are as follows:

```
    // 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
("bankMarker:bankServer", hostname);
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.

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 Orbix Java exception handling. Refer to the chapter "Exception Handling" on page 143 for details of how to handle exceptions in Orbix Java.

This example 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 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 Orbix Java, 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 252.

The CORBA Approach to Using the DII

This section demonstrates how to use the DII using the Orbix Java 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 241. 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 the chapter "Making Objects Available in Orbix Java" on page 127 for further information on this topic.

Note: In previous versions of Orbix Java, when using the DII, object references were associated with the default ORB (`_CORBA.Orbix`). Now, the object references are associated with the ORB in context. The enables multiple ORB support.

Creating a Request

CORBA specifies two ways to construct a `Request` object. These are implemented in Orbix Java 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 251.

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 Orbix Java 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 the chapter "Type any" on page 229, 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 250.

   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()creates a NamedValue and adds it to the NVList. It returns a NamedValue pseudo object reference for the newly created NamedValue.

   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 initialized 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 245 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 Orbix Java 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`, initialized 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 249. 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 for Using the DII" 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 Orbix Java 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, Orbix Java 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, Orbix Java 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

Orbix Java 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 291. All filter points that you implement are called during the invocation of a dynamic request.

The Interface Repository

This chapter describes the Interface Repository (IFR). This is the Orbix Java component that provides persistent storage of IDL interfaces, modules, and other IDL types. Orbix Java programs can query the Interface Repository at runtime to obtain information about IDL definitions.

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 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.

Orbix Java 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 Orbix Java 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 Orbix Java installation. The overall requirements for using the Interface Repository are as follows:

- You must set the `IT_INT_REP_PATH` in the Orbix Java 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 256.
- An application must import relevant Java classes.

Installing the Interface Repository

The Interface Repository is itself an Orbix Java 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 Orbix Java 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:

```
    putitj 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.

You can use the `putidl`, `readifr` and `rmidl` utility commands to access the Interface Repository, Refer to the ***Orbix Administrator’s Guide Java Edition*** for details.

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.

EnumDef	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 260 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 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 26.

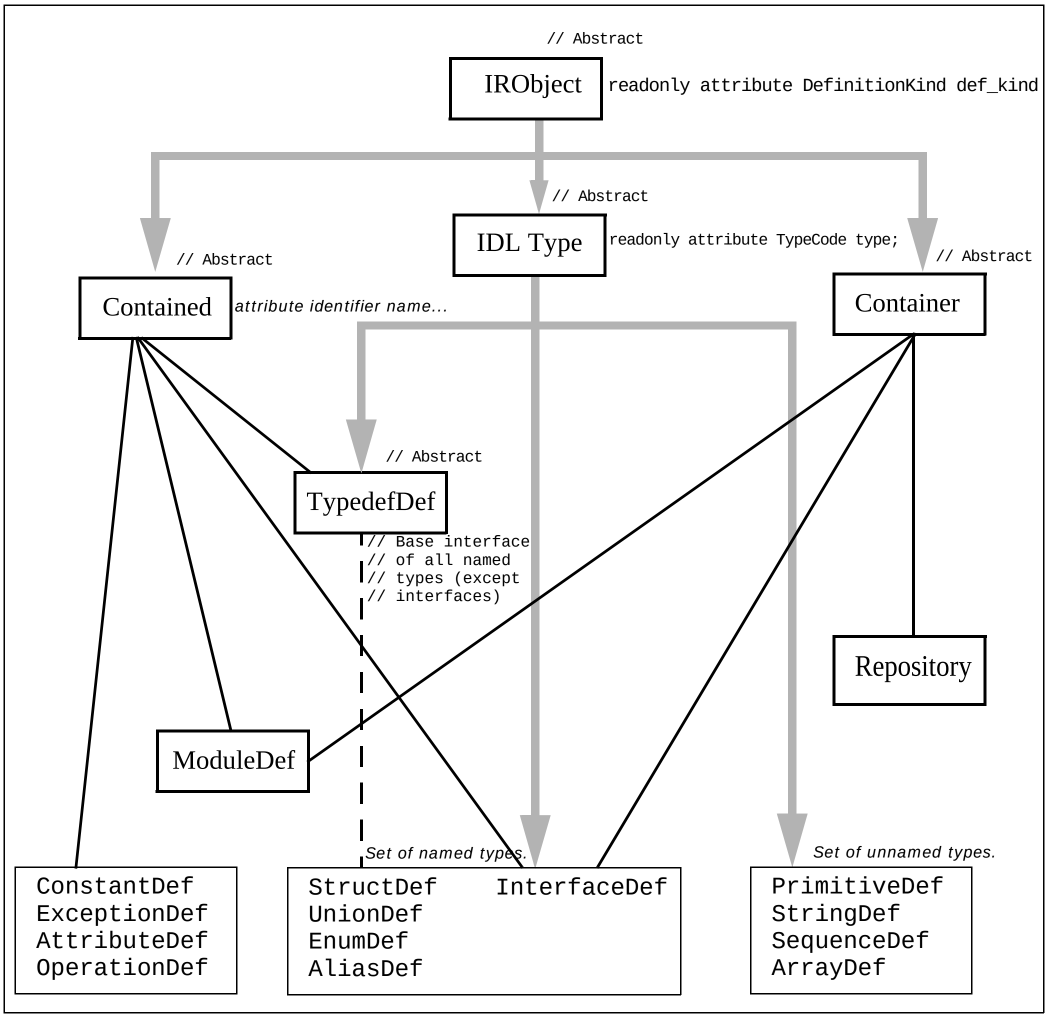


Figure 26: *Hierarchy for Interface Repository Interfaces*

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.

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 *Orbix Java Edition 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 272.

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 264. 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 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 ***Orbix Programmer's Reference Java Edition*** .

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 272 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 256.

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 Orbix Java `bind()` function, as follows:

```
import org.omg.CORBA.Repository;
import org.omg.CORBA.RepositoryHelper;
Repository repVar =
    RepositoryHelper.bind
    ("IDL\\iona.com/Repository:IFR", "hostname");
```

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 Orbix Java 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 ***Orbix Programmer's Reference Java Edition*** 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
    ("IDL\\iona.com/Repository:IFR", “hostname”);
    //
    //  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 `do it`. 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 Orbix Java.

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 Orbix Java 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 {
    interface account {
        ...
    };
     #pragma version account 2.5
};
```

yield the following Repository IDs:

```
IDL:finance:1.0
and
IDL:finance/account:2.5
```

It is important to realize 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.

Service Contexts

Service contexts provide a means of passing service-specific information as part of IIOP message headers. This chapter describes Orbix Java 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 in IDL 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 Orbix Java can only be used over IIOP.

The Orbix Java Service Context API

The Orbix Java API for service contexts comprises the following:

- Service context handlers.
- Service context lists.
- ORB interfaces.

Service Context Handlers

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 280 for more details.

The ServiceContextHandler base class has the following structure:

```
// Java

public abstract class  ServiceContextHandler {

    // Fields
    public int m_serviceContextId;
    public Object m_serviceContextObject;

    // Constructors
    public ServiceContextHandler(int);

    // Methods
    public int _getID();
    public Object _getObject();
    public void _setObject(Object);
    public abstract boolean incomingReplyHandler(Request);
    public abstract boolean incomingRequestHandler(Request);
    public abstract boolean outboundReplyHandler(Request);
    public abstract boolean outboundRequestHandler(Request);
}
```

Service Context Lists

A ServiceContextList is a field in an 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.

The ServiceContextList class has the following structure:

```
    public class  ServiceContextList {
        // Constructors
        public ServiceContextList();
        public ServiceContextList(ServiceContext[]);

        // Methods
        public void add(int, byte[]);
        public final ServiceContext get(int);
        public final ServiceContext[] getList();
        public void register(ServiceContext);
        public final int size();
    }
```

ORB Interfaces

Orbix Java provides APIs in class IE.Iona.OrbixWeb.CORBA.ORB to allow you to enable service contexts and to register service context handlers with the ORB.

Enabling Service Contexts

The following ORB API allows you to enable ServiceContexts on the ORB:

```
    public void enableServiceContextList(boolean state);
```

You must call this method to use service contexts.

Registering Service Context Handlers

The following ORB APIs allow you to register the service context handler with the ORB:

```
// Java

public class ORB {
    ...
    public void registerPerRequestServiceContext
                    (ServiceContextHandler CtxHandler);

    public void unregisterPerRequestServiceContext
                                    (int CtxHandlerId);

    public void registerPerObjectServiceContext
                (ServiceContextHandler CtxHandler,
                org.omg.CORBA.Object HandledObject);

    public void unregisterPerObjectServiceContext
                    (int CtxHandlerId)
}
```

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.

Using Service Contexts in Orbix Java Applications

Service contexts in Orbix Java 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 Orbix Java applications.

Client Side

To add service context information to all requests leaving a client application, perform the following steps:

1. Call the `enableServiceContextList()` 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 `SrvCntxtId`.
4. Register this handler instance with the ORB using the following method:

   ```
   void registerPerRequestServiceContextHandler
                  (ServiceContextHandler CtxHandlerId)
   ```

   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, perform the following steps:

1. Call the `enableServiceContextList()` 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 `SrvCntxtId`. 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
   CtxHandlerId);
   ```

   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 required 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.

Per-Request ServiceContextHandler Example

The service context example in this section sends a String message using a per-request `ServiceContextHandler`.

First, you should place the following code in both your client and your server applications:

```
//java
IE.Iona.OrbixWeb._OrbixWeb.ORB
(orb).enableServiceContextList(true);

mySrvContext = new myServiceContextHandler(5);

IE.Iona.OrbixWeb._OrbixWeb.ORB
(orb).registerPerRequestServiceContextHandler
                                (mySrvContext);
```

Second, you can use the following example code to send a String using a per-request `ServiceContextHandler`:

```
//java
import IE.Iona.OrbixWeb.Features.ServiceContextHandler;
import IE.Iona.OrbixWeb.Features.ServiceContext;
import IE.Iona.OrbixWeb.CORBA.Request;
import IE.Iona.OrbixWeb.CORBA.*;
import IE.Iona.OrbixWeb.*;
import IE.Iona.OrbixWeb.CORBA.Any;
import IE.Iona.OrbixWeb.CORBA.ORB;

public class myServiceContextHandler
    extends ServiceContextHandler {

    long num = 0;

    // constructor
    public myServiceContextHandler(int id) {
        super(id);
        System.out.println
            ("Created ServiceContextHandler");
    }

    public boolean outboundReplyHandler(Request req) {
        return true;}
    public boolean outboundRequestHandler(Request req)
        {
        String str = "hello world";
        System.out.println("Add Service Context list to
            Request \n" + "\ttarget \t" + req.target()
            +"\tcalling \t" + req.operation() );

        IE.Iona.OrbixWeb.CORBA.Any a = new
            IE.Iona.OrbixWeb.CORBA.Any
                (_CORBA.IT_INTEROPERABLE_OR_KIND);

        a.insert_string(str);
        ServiceContext sc = new ServiceContext();
        sc.context_id = _getID();
        sc.context_data = str.getBytes();

        String str2 = new String(sc.context_data);
        /*Byte b = null;
      for (int i=0; i < sc.context_data.length; i ++) {
            str2 += Byte.toString(sc.context_data[i]);
        } */
        System.out.println("converted to = " + str2);
        req.addServiceContext(sc);

        return true;
    }
    public boolean incomingReplyHandler(Request req) {
        return true;}
    public boolean incomingRequestHandler(Request req)
         {
        String str = null;
        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());

        IE.Iona.OrbixWeb.CORBA.Any a =
            new IE.Iona.OrbixWeb.CORBA.Any
                (_CORBA.IT_INTEROPERABLE_OR_KIND);

        a.insert_string(str);
        str = new String(sc.context_data);
        System.out.println("Extracted from Request \n" +
        "\tID \t\t" + sc.context_id + " " + str);

        return true;
    }
}
```

ServiceContext Per-Object Model

This section gives an overview of implementing per-object service contexts in Orbix Java 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.

Service Context 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 recognized 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 recognized 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 Orbix Java filter points. In Orbix Java, there are ten filter points, including the in reply and out reply failure filter points. Refer to "Filters"for more details. The service context mechanism provides four more points for interaction with requests and replies in a typical invocation.

Figure 27 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()`.

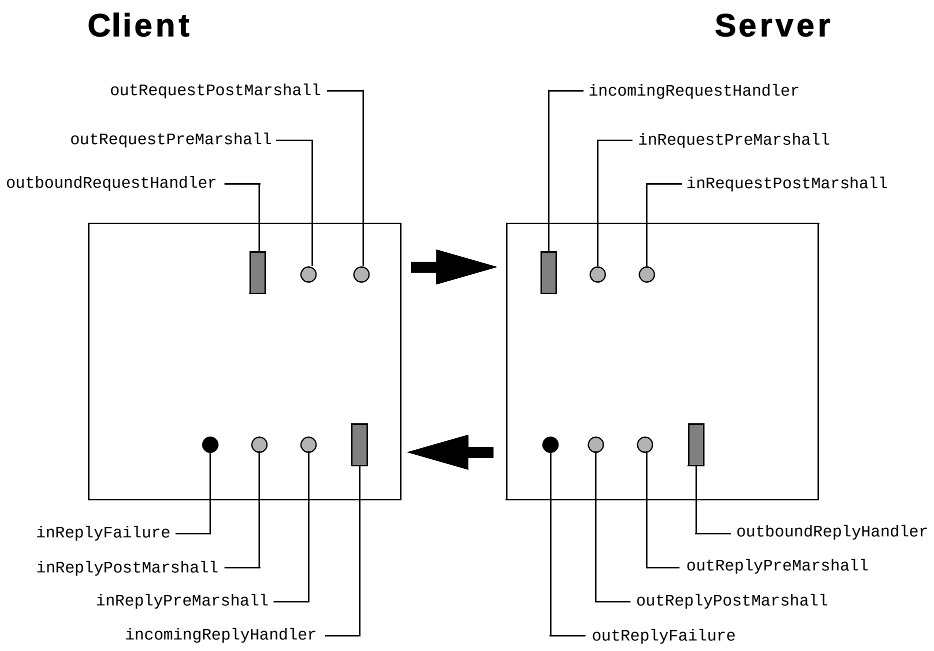


Figure 27: *ServiceContext Handlers and Filter Points*

Part V

Advanced Orbix Java Programming

In this part

This part contains the following:

Filters

Orbix Java 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 Orbix Java:

- *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 28 on page 291. This chapter briefly introduces each filter type, and then describes each in detail.

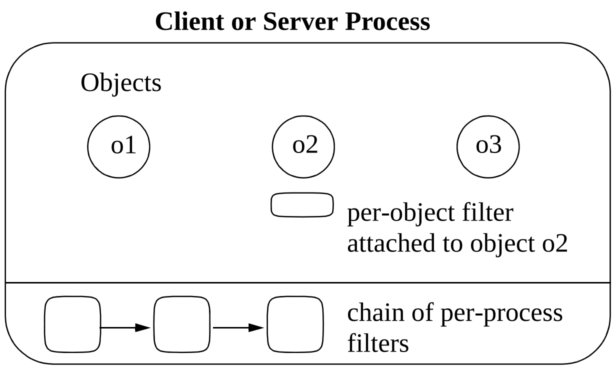

Client or Server Process
Objects
o1
o2
o3
per-object filter attached to object o2
chain of per-process filters

Figure 28: *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. Orbix Java allows you to associate filters with a particular ORB instance.

By default, Orbix Java 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 ***Orbix Programmer's Reference Java Edition*** for details of `org.omg.CORBA.ORB.init()` and class `IE.Iona.OrbixWeb.Features.Filter`.

Orbix Java also provides constructors that associate a `ThreadFilter` or an `AuthenticationFilter` with a particular ORB instance. Refer to package `IE.Iona.OrbixWeb.Features` in the ***Orbix Programmer's Reference Java Edition*** 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 29 on page 294. 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 29 on page 294.

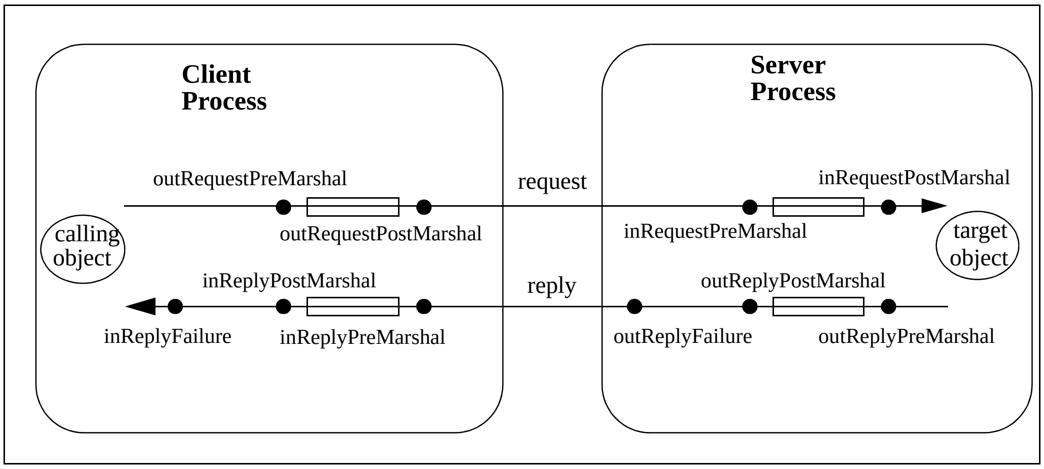


Figure 29: *Per-Process Monitor Points*

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 Orbix Java 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 303 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 realize 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 305.

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 Orbix Java 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 ***Orbix Programmer's Reference Java Edition*** 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 297 for 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 ***Orbix Programmer's Reference Java Edition***.

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" for more details. Modification of data inserted by the Orbix Java 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 ***Orbix Programmer's Reference Java Edition***.

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 Orbix Java 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 ***Orbix Programmer's Reference Java Edition***.

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 Orbix Java 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 Orbix Java daemon. The Orbix Java 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 Orbix Java 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).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. Orbix Java 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 behavior, the server-side authentication filter point inRequestPreMarshal() must be appropriately overridden in all servers with which the client communicates.

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 Orbix Java.

Multi-Threaded Clients and Servers

Normally, Orbix Java 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 synchronization 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 Orbix Java 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 synchronization 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 Orbix Java

Orbix Java supports multi-threaded Java servers that handle multiple client requests. The Java language is multi-threaded and the Orbix Java runtime is thread-safe.

Using Class ThreadFilter

The class `IE.Iona.OrbixWeb.Features.ThreadFilter` enables the implementation of custom threading and queuing policies in Orbix Java.

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 behavior 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 ***Orbix Programmer's Reference Java Edition*** for more details on `IE.Iona.OrbixWeb.Features.ThreadFilter`.

Models of Threading

The following are the three models of thread support provided by Orbix Java:

- *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 Orbix Java

This section gives a brief description of how these models can be implemented in Orbix Java.

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 in the `Threads` demonstrations in the `demos/orbixjava/` directory of your Orbix Java installation.

Smart Proxies

Smart proxies are an Orbix Java-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 Orbix Java manages proxies.

Proxy Classes

For each IDL interface, the Orbix Java 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. Orbix Java implicitly constructs a standard proxy when an object reference enters the client address space. Experienced Orbix developers should note that Orbix Java does not use proxy factory classes to construct standard proxy objects. However, Orbix Java 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 Orbix Java. 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. Orbix Java 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, Orbix Java 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. Orbix Java 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 Orbix Java `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, Orbix Java 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, Orbix Java 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. Orbix Java allows you to associate smart proxies with particular ORB instances.

By default, Orbix Java 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 ***Orbix Programmer's Reference Java Edition*** 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.

Using Smart Proxies

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 Orbix Java server process.

It is also assumed that an Orbix Java 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 initializes 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 30.

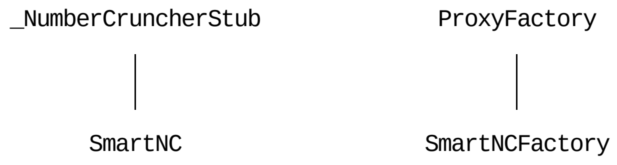


Figure 30: *Class Hierarchy for Smart Proxy Classes*

2. The `SmartNCFactory.New()` method is called by Orbix Java to signal that a smart proxy can be created. Orbix Java 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.

Loaders

This chapter describes the use of loaders, an Orbix Java-specific feature designed to support persistent objects.

When an operation invocation arrives at a server process, Orbix Java searches for the target object in the internal object table for the process. By default, if the object is not found, Orbix Java returns an exception to the caller. However, if one or more *loader* objects are installed in the process, Orbix Java informs the loader about the *object fault* and allows it to load the target object and resume the invocation transparently to the caller. Orbix Java 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, Orbix Java 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()`

  Orbix Java 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, Orbix Java 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 Orbix Java-specific `ObjectRef` interface.

- `record()` and `rename()`

  These methods are used to control naming of objects, and they are explained in the chapter "Making Objects Available in Orbix Java".

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 Orbix Java.

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. Orbix Java allows you to associate loaders with particular ORB instances.

By default, Orbix Java 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 ***Orbix Programmer's Reference Java Edition*** for more details on class `LoaderClass`.

Refer to the section "Example Loader" on page 324 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. Orbix Java informs the loader object when the object is named, renamed or saved. If an object does not have a specified loader, Orbix Java 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);
                        ...
    }
}
```

Orbix Java 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, Orbix Java calls the object's loader to confirm the chosen name, thus allowing the loader to override the choice. In the first two cases above, Orbix Java calls `record()`; in the last case it calls `rename()` because the object already exists.

Orbix Java executes the following algorithm when an object is created, or an object's existing marker is changed:

- If the specified marker is not null, Orbix Java 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, Orbix Java 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, Orbix Java passes a null value to the loader (by calling `record()` or `rename()`) which must then choose a name. Orbix Java 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, Orbix Java 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, Orbix Java 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\orbixjava\loaders_per_simp directory of your Orbix Java 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:

```
idlj -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 Orbix Java 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 (b1234: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 recognizes 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.

Opaque Types

Orbix Java 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 Orbix Java.

In accordance with the CORBA standard, Orbix Java objects are passed to and from IDL operations *by reference*. Orbix Java 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 that implements the opaque object.
- The opaque's Helper class that implements the stream-based marshalling and unmarshalling of the opaque object.

Possible Alternative Solutions

As outlined in the previous section, the Orbix 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.

Note: Because of the Orbix Java-specific nature of opaques, you cannot use opaque types with the CORBA-defined Interface Repository.

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/orbixjava/Date` directory of your Orbix Java 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:

```
idlj -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/orbixjava/Date` directory of your Orbix Java installation for an example client/server application that uses the `Date` type.

Transforming Requests

This chapter describes how you can modify the data buffers containing Orbix Java operation call information immediately before and after transmission across the network.

In Orbix Java, 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 Orbix Java `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 Orbix Java 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 ***Orbix Programmer's Reference Java Edition*** for full details of the IT_reqTransformer class.

The transform() method is called by Orbix Java 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 Orbix Java 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

Orbix Java 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 Orbix Java 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 Orbix Java 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 `Request`s 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");
```

Part VI
Appendix

In this part

This part contains the following:

IDL Compiler Switches

This appendix describes the command-line switches to the IDL Compiler.

The IDL Compiler supports the following switches to the `idlj` command:

`-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 Orbix Java IDL pre-processor. Do not pass the output of the pre-processor to the Orbix Java IDL compiler, but output the pre-processed file to standard output. By default, the output of the Orbix Java IDL pre-processor is sent to the Orbix Java 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 Orbix Java -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 that 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.
`-juATC`	Creates an alias TypeCode for the specified file. This contains the TypeCode's Repository ID, name and original type. Alias Typecodes are required for ORB interoperability.
`-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: You must process each IDL file through the IDL compiler. Including an IDL file in another (using `#include`) does not produce output for the included file (unless the `-N` switch is specified to the compiler). Otherwise, Java code generation occurs more than once for a file that is included in more than one file.

Index