Programmer's Guide

Netscape Internet Service Broker for Java

Version 1.0

Version 1.0

Printed in USA
99 98 97 10 9 8 7 6 5 4 3 2 1

Netscape Communications Corporation 501 East Middlefield Road, Mountain View, CA 94043

Contents

Preface

Netscape Internet Service Broker for Java (ISB for Java) allows you to develop and deploy applications that use distributed objects, as defined in the Common Object Request Broker Architecture (CORBA) specification. This guide provides information about developing distributed object-based applications.

Overview

This preface lists the contents of this guide, describes typographical and syntax conventions used throughout the guide, and provides references for more information about CORBA.

- Organization of this Guide
- Typographic Conventions
- Platform Conventions
- Where to Find Additional Information

Organization of this Guide

This guide includes the following chapters:

- ISB for Java Basics introduces ISB for Java, a complete implementation of the CORBA 2.0 specification for developing distributed object-based applets. It provides a brief description of features and some information about upgrading from a previous release of ISB for Java.
- Getting Started describes the development of distributed, object-based applications with ISB for Java. A sample application illustrates each step of the development process. It is used throughout this guide as new features are introduced.

- Naming and Binding to Objects explains how objects are named and how to locate objects using Uniform Resource Locators.
- Object and Implementation Activation describes how object servers are implemented and how objects are made available for use by client applications.
- The Dynamic Invocation Interface explains how client applications can dynamically construct and invoke operation requests.
- The Tie Mechanism describes how to create Java interfaces without inheriting from `CORBA.Object`.
- The Interface Repository explains how to use the interface repository to dynamically obtain objects and their interfaces.
- Defining CORBA Interfaces With Java describes how you can generate client stubs and server skeletons using existing Java code instead of IDL.
- Managing Threads and Connections describes the use of threads and thread policy in client applications and object implementations. This chapter will help you choose the best thread and connection manager for your application.
- The Dynamic Skeleton Interface describes how object servers can dynamically create object implementations at run time to service client requests.
- Troubleshooting explains how to troubleshoot common build and compilation errors, how to debug the runtime, and includes a section describing properties and variables.

Typographic Conventions

This guide uses the following conventions:

Convention	Used for
boldface	Boldtype indicates that syntax should be typed exactly as shown. For UNIX, it indicates database names, filenames, and similar terms.
italics	Italics indicates information that the user or application provides, such as variables in syntax diagrams. It is also used to introduce new terms.
`computer`	Computer typeface is used for sample command lines and code.
UPPER CASE	Uppercase letters indicate Windows filenames.

[ ]	Brackets indicate optional items.
...	An ellipsis indicates that the previous argument can be repeated.
|	A vertical bar separates two mutually exclusive choices.
. . .	A column of three dots indicates the continuation of previous lines of code.

Platform Conventions

This guide uses the following symbols to indicate that information is platform-specific:

W All Windows platforms, including Windows 3.1, Windows NT, and Windows 95

NT Windows NT only

95 Windows 95 only

U All UNIX platforms

Where to Find Additional Information

For more information about ISB for Java, see the following source:

- *Netscape Internet Service Broker for Java Reference Guide* This guide contains information on the ISB commands and Java interfaces.

For more information about the CORBA specification, see the following source:

- *The CORBA 2.0 Specification* -96-03-04. This document is available from the Object Management Group at http://www.omg.org and describes the architectural details of CORBA.

For more information about programming with Java and CORBA, see the following sources:

- *Client/Server Programming with Java and CORBA* by Robert Orfali and Dan Harkey.
- *Java Programming with CORBA* by Andreas Vogel and Keith Duddy.

- *Instant CORBA* by Robert Orfali, Dan Harkey, and Jeri Edwards.

Chapter

1

ISB for Java Basics

This chapter presents Netscape Internet Service Broker for Java (ISB for Java), an implementation of the CORBA 2.0 specification for developing distributed object-based applications. It includes the following major sections:

- What is CORBA?
- What is ISB for Java?
- Overview of the Development Process
- Interoperability with ISB for C++
- ISB for Java Features

What is CORBA?

The Common Object Request Broker Architecture (CORBA) specification was developed by the Object Management Group to address the complexity and high cost of developing software applications. CORBA defines an object-oriented approach to creating software components that can be reused and shared between applications. Each object encapsulates the details of its inner workings and presents a well-defined interface, which reduces application complexity. The cost of developing applications is also reduced because once an object is implemented and tested, it can be used over and over again.

ISB for Java's Object Request Broker (ORB) connects a *client* (either an application running in a Java virtual machine or an applet running in a Java-enabled browser) with the objects (called *services* or *servers*) it wishes to use. The client does not need to know whether the service resides on the same computer or on a remote computer somewhere on the network. The client only needs to know the service's name and understand how to use the service's interface. The ORB locates the object, routes the request, and returns the result. You can develop objects that act both as clients (using services from other objects) and servers (providing services to other objects).

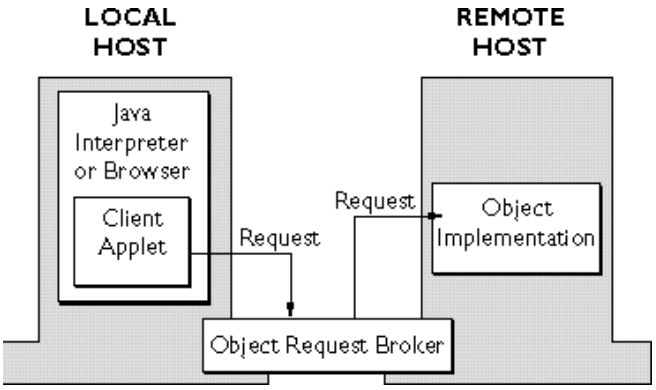

Figure 1.1 Client application acting on an object through an ORB.

Note: The ORB is not a separate process. It is a collection of libraries and other resources that clients and services use to communicate.

What is ISB for Java?

ISB for Java is an ORB that provides a complete implementation of the CORBA specification. ISB for Java makes it easy for you to develop distributed, object-based clients and services. ISB for Java offers these important features:

- Support for the Java programming language.
- Object lookup by name or by URL.
- The ability to distribute objects across a network.
- Support for persistent object references.

- Interoperability with other ORB implementations.

Overview of the Development Process

The first step to creating an application with ISB for Java is to specify all of your service's interfaces using the OMG's Interface Definition Language (IDL). The IDL mappings for the Java language are covered in the *Netscape Internet Service Broker for Java Reference Guide*. The interface specification you create is used by the `idl2java` compiler to generate stub routines for the client application or applet and skeleton code for the service implementation. The stub routines are used by the client for all method invocations. You use the skeleton code, along with code you write, to implement the service. The completed code for the client and service is used as input to your Java compiler to produce a Java application or applet and an object service.

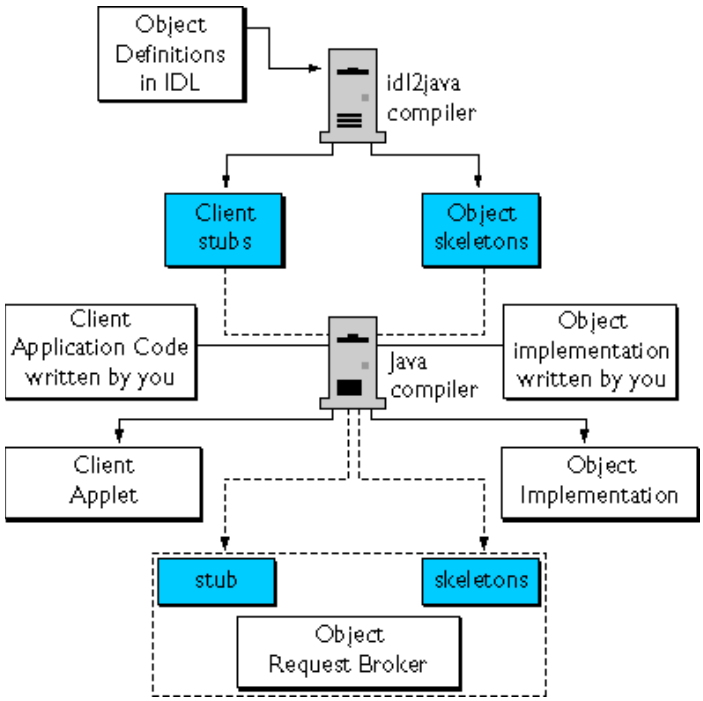

Figure 1.2 Creating an application and a service using ISB for Java.

Interoperability with ISB for C++

Java clients created with ISB for Java can communicate with services developed with ISB for C++, a separate product. Use the same IDL code you used to develop your Java application as input to the ISB IDL compiler supplied with ISB for C++. Then use the resulting C++ skeletons to develop the service implementation. Also, services written using ISB for Java will work with clients written using ISB for C++. In fact, a service written using ISB for Java will work with *any* CORBA-compliant client; a client written using ISB for Java will work with *any* CORBA-compliant service.

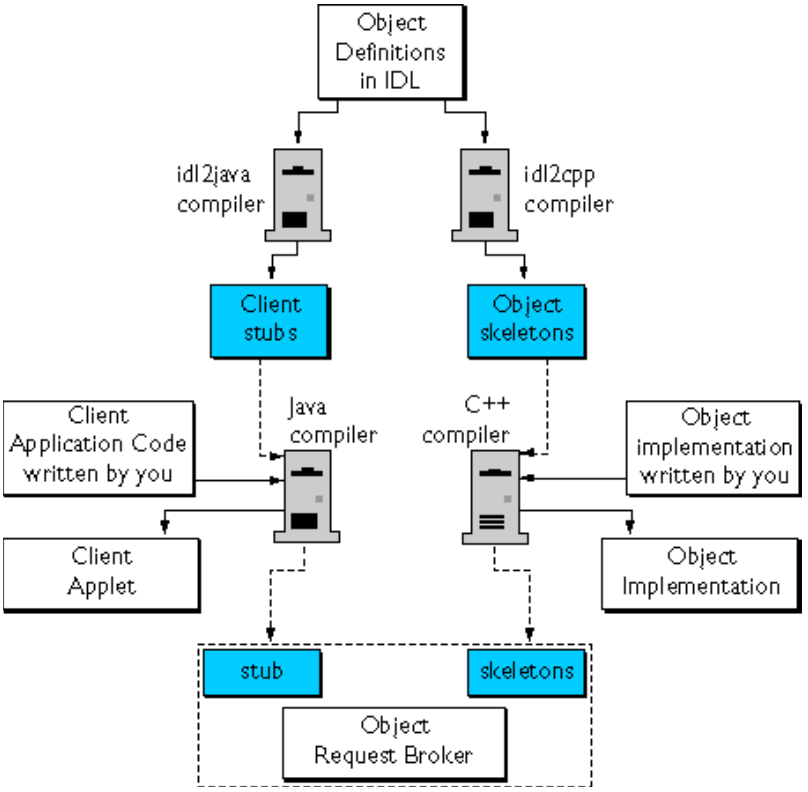

Figure 1.3 Creating a client with ISB for Java and a service with ISB for C++.

ISB for Java Features

In addition to providing the features defined in the CORBA specification, ISB for Java offers enhancements that increase application performance and reliability.

Feature	Description
Dynamic Invocation Interface	A client can obtain a service's interface and dynamically construct requests to invoke the methods it provides. See The Dynamic Invocation Interface for more details.
Interface Repository	The interface repository (IR) maintains information about ORB objects, such as modules, interfaces, operations, attributes, and exceptions. An IDL interface is provided that enables clients to query the IR to obtain language binding information or to discover newly added interfaces. The ORB also accesses the IR when it needs to check the types of values in a client request or verify an interface inheritance graph. For more information, see The Interface Repository.
Web Naming	Web Naming allows you to associate Uniform Resource Locators with objects, allowing an object reference to be obtained by specifying a URL. For more information, see Naming and Binding to Objects.
Caffeine: Defining Interfaces Without IDL	ISB for Java's `java2iiop` utility (one in a set of tools collectively called *Caffeine*) allows you to use Java instead of IDL to define interfaces. You can use the `java2iiop` utility to adapt existing Java code to use distributed objects, or if you do not have the time to learn IDL. The `java2iiop` utility generates the necessary container classes, client stubs, and server skeletons from Java code. For more information, see Defining CORBA Interfaces With Java.
Enhanced Thread and Connection Management	ISB supports a *thread-per-session* policy for managing connections between client applications and servers. For more information, see Managing Threads and Connections.

Feature	Description
Dynamic Skeleton Interface	The Dynamic Skeleton Interface (DSI) provides a mechanism for creating an object implementation that does not inherit from a generated skeleton interface. Normally, an object implementation is derived from a skeleton class generated by the `idl2java` compiler. DSI allows an object to register itself with the ORB, receive operation requests from a client, process the requests, and return the results to the client without inheriting from a skeleton class. For more information, see The Dynamic Skeleton Interface.
IDL to Java Mapping	ISB for Java conforms with the *OMG Java Language Mapping RFP*. See "IDL to Java Mapping" in the *Netscape Internet Service Broker for Java Reference Guide* for a summary of ISB's current IDL-to-Java language mapping as implemented by the `idl2java` compiler. For each IDL construct, a section describes the corresponding Java construct.

Chapter

2

Getting Started

This chapter steps you through setting up your development environment, helps you become acquainted with some of the development tools, and describes how to develop a distributed, object-based application with ISB for Java. A sample application illustrates each step of the process. This chapter includes the following major sections:

- Setting Up
- Developing Applications
- A Sample Application

Setting Up

To develop applications using ISB for Java, you must have:

- JDK 1.1 or higher
- Enterprise Server 3.0f, 3.01, or higher with WAI-Patch #P85416

Before you start, set up your development environment and the Enterprise Server, and set your PATH and CLASSPATH environment variables.

Enterprise Server Requirements

To use CORBA, WAI-Management must be ON.

Applications developed using ISB for Java require access to an Enterprise Server directory. More specifically, the server must enable Web Publishing and the directory must allow HTTP PUTs. One way to do this is to use the Administration Server to set up an additional directory to enable write access. The examples in this chapter were developed using the directory `c:\projects` mapped as an additional directory on the server named `myServer.nscp.com/projects` with write access enabled. For details on enabling Web Publishing and setting up an additional directory, consult the Enterprise Server documentation.

Web Naming (Naming Service) security must be configured to give write access to the `iiopnameservice` configuration style.

More details about setting up your Enterprise Server are available at `http://developer.netscape.com/library/technote/components/corba/webnaming/webnaming.htm`.

Setting Your PATH Variable

Set your PATH environment variable to point to the directory where the JDK Java binaries have been installed and to the `bin` directory of the ISB distribution.

95 Assuming that both the JDK and the ISB distribution were installed in `c:\` using default settings, you can set your path with the following DOS command:

```
prompt> set PATH=c:\jdk1.1.2\bin;c:\netscape\suitespot\wai\bin;%PATH%
```

NT For NT 3.51, you can use the DOS `set` command to set environment variables, but it is easier to use the System Control Panel. Assuming that both the ISB and Java distributions were installed in `c:\` using default settings, use the Control Panel to start the System program, enter `PATH` as the variable to edit and add the following directories to the path:

```
c:\jdk1.1.2\bin;c:\netscape\suitespot\wai\bin;
```

For NT 4.0, use the Control Panel to start the System program and select the Environment tab from the System Properties window. Select the PATH variable from the User Variables list, then add the following to the path:

```
c:\jdk1.1.2\bin;c:\netscape\suitespot\wai\bin;
```

Click the Set button to accept the new path information.

U Assuming that you installed ISB and Java in `/usr/local` using default settings, update your `PATH` environment variable as follows:

```
prompt> setenv PATH /usr/local/netscape/suitespot/wai/bin:/usr/local/jdk1.1.2/bin:$PATH
```

Setting Your CLASSPATH Variable

The `CLASSPATH` environment variable tells the Java interpreter where to look for classes. Add JDK, Enterprise Server, and WAI classes to your CLASSPATH. Note that the Communicator classes in `java40.jar` are needed *only* if you are developing a service applet that will run in the Communicator. Typically, services run as stand-alone applications, so classes in `java40.jar` are *not* needed.

W You could save the following commands in a batch file to set CLASSPATH for ISB development. You may need to edit paths to match your directory structure.

```
          rem JDK classes.
set classpath=.;c:\jdk1.1.3\lib\classes.zip;
          rem ES classes and WAI.
set
classpath=%classpath%c:\netscape\suitespot\wai\java\nisb.zip;c:\netscape\suitespot\wai\jav
a\wai.zip;
          rem Communicator Java classes (to import netscape.security.PrivilegeManager).
          rem These classes are required only if you developing a service that will run
          rem in the Communicator.
          rem They are not needed for services running in the Enterprise Server.
          rem Note: To use the enablePrivilege call, the applet must be delivered
          rem in a signed jar file, or the user preference for codebase principal
          rem support must be enabled.
set classpath=%classpath%c:\proga~1\netscape\communicator\program\java\classes\java40.jar;
```

Developing Applications

To develop distributed applications with ISB for Java, you must first identify the objects required by the application. You will then usually follow these steps:

1. Write a specification for each object using the Interface Definition Language (IDL).
2. Use `idl2java` to generate the client stub code and service skeleton code.
3. Implement the service interface.

4. Write an application (often called a *server application* or *object server*) to create, activate, and register an instance of the service.

5. Write the client application (or applet) code.

6. Use `javac` to compile the service interface, server application, and client application (or applet).

7. Start the server application.

8. Run the client application (or applet). When the client invokes a service method, the call goes through the client-side stubs to the ORB, which uses the server-side skeletons and information from the server application to locate the service implementation and call the method.

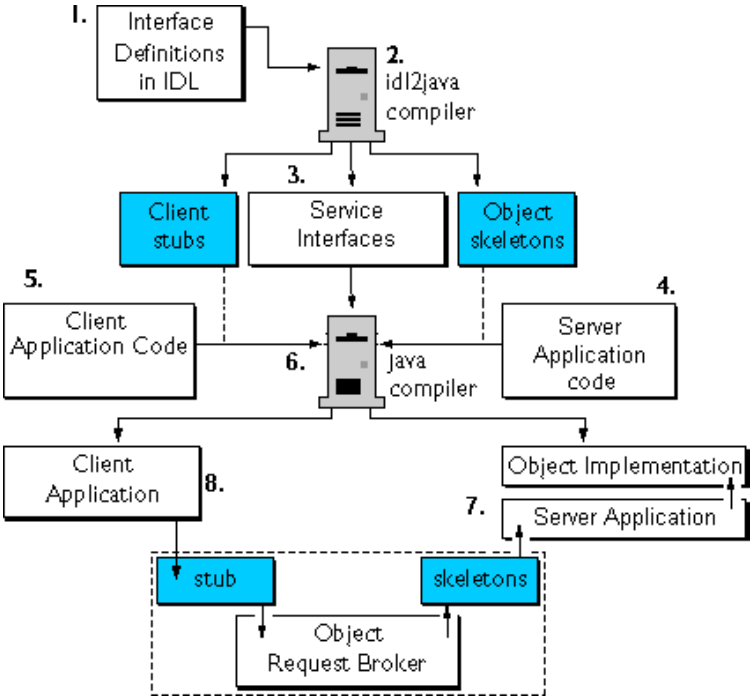

Figure 2.1 Developing and Running an ISB application.

A Sample Application

The rest of this chapter presents a small sample application. A Java application named MsgService exposes one service, accessed via the Hello interface, that prints a message in the server window and the client window (when you run the client as an application) or the Java console (when you run it as an applet). The following figure shows the MsgService application running in a DOS window, the MsgClient application running in another DOS window, the MsgClient applet running in the Communicator, and a message displayed in the Java Console.

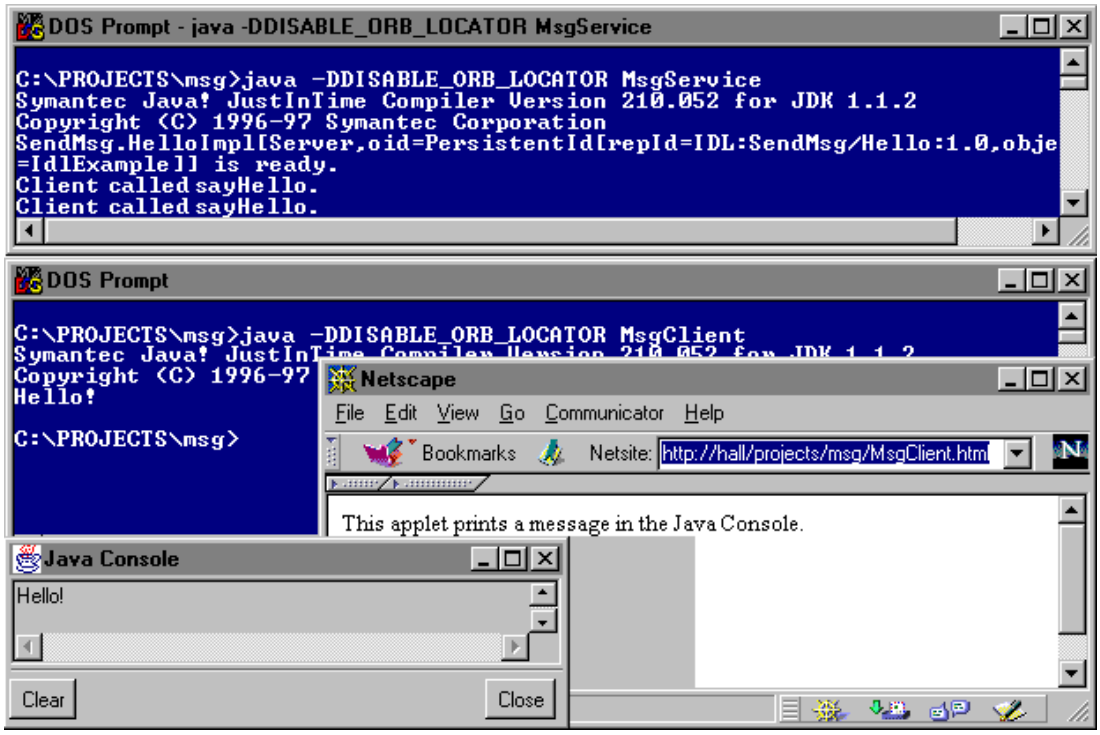

Figure 2.2 MsgService and MsgClient windows.

This sample application was developed under Windows NT 4.0. It assumes that `c:\projects\msg` is its root directory, that some files reside there, some in a subdirectory named `SendMsg`. If you recreate this sample application using a different configuration, change code and commands accordingly.

The most important files in this application are listed in the following table. You, the programmer, will write or edit each of these files. The application also uses files generated by the `idl2java` compiler.

Filename	Description
SendMsg.idl	Defines the interface to the Hello service. Resides in the application root directory.
HelloImpl.java	Implements the Hello interface. Resides in the SendMsg subdirectory of the application root directory.
MsgService.java	Activates and registers the service, making it available to clients. Resides in the application root directory.
MsgClient.java	Accesses the service and calls a method from its interface. Can be run as an application or an applet. Resides in the application root directory.
MsgClient.html	A page you can display in the Communicator or the AppletViewer to run the client applet. Resides in the application root directory.

Following are the steps to develop this sample application.

1. Defining Interfaces in IDL
2. Running the idl2java Compiler
3. Implementing Interfaces
4. Writing a Server Application
5. Writing a Client Application or Applet
6. (Using `javac` to compile the interfaces, server application, and client application or applet is covered as part of the preceding three steps.)
7. Starting a Server Application
8. Starting a Client Application

Defining Interfaces in IDL

The standard way to define CORBA interfaces is to use the Interface Definition Language (IDL). For details about IDL, see "The IDL to Java Mapping" in the *Netscape Internet Service Broker for Java Reference Guide*. The following file (named `SendMsg.idl`) defines the SendMsg module that defines the Hello interface. The interface provides one method, `sayHello`, that returns a string.

Listing 2.1 SendMsg.idl defines the Hello interface.

```
// SendMsg.idl
module SendMsg {
    interface Hello {
        string sayHello();
    };
};
```

An IDL file begins by defining a *module* (which corresponds to a Java package). A module contains one or more interfaces that provide one or more methods. (ISB for Java provides additional tools and utilities, collectively known as Caffeine, that you can use to define interfaces in Java without using IDL.)

Running the idl2java Compiler

The ISB for Java IDL compiler is named `idl2java` and, as its name implies, generates Java code from a file of IDL specifications. The following command compiles the `SendMsg.idl` file.

```
c:\projects\msg> idl2java -no_tie -no_comments SendMsg.idl
```

The `-no_tie` flag tells the compiler not to generate files to support the tie mechanism (which is not used in this example), and the `-no_comments` flag tells the compiler not to put comments in the files it generates (which makes the files easier to read). For more information on the command line options for the `idl2java` compiler, see "Commands" in the *Netscape Internet Service Broker for Java Reference Guide*.

Generated Files

Because Java allows only one public interface or class per file, compiling `SendMsg.idl` generates several `.java` files. The `idl2java` compiler creates a subdirectory named `SendMsg` (which is the module name specified in the IDL file) and stores generated files there. IDL modules are mapped to Java packages, so all of the files listed below are part of the `SendMsg` package.

Filename	Description
Hello.java	The Hello interface declaration in Java.
_example_Hello.java	Provides a code framework you can fill into implement the Hello interface. After adding code to an example file, save your work using a different filename. This sample application uses the name `HelloImpl.java`.
_sk_Hello.java	Skeleton code for the Hello interface implementation on the server side. This file is part of the SendMsg package. It contains generated code for the `Hello` service object on the server side. Do *not* modify this file.
_st_Hello.java	Stub code for the Hello interface implementation on the client side. This file is part of the SendMsg package. It contains generated code for the `Hello` service object on the client side. Do *not* modify this file.
HelloHolder.java	This class provides a holder for passing method parameters.
HelloHelper.java	This class defines helpful utility functions.

For more information about the Helper and Holder classes, see "Generated Classes" in the *Netscape Internet Service Broker for Java Reference Guide*.

Implementing Interfaces

In CORBA, interface definitions and implementations are separate and distinct. You define interfaces once, then implement them one or more times in one or more languages, depending on your application requirements. In this sample application, the service interface is implemented in Java using a file generated by `idl2java`.

When the `idl2java` compiler processes the file `SendMsg.idl`, it generates files named `Hello.java` and `_example_Hello.java. The file Hello.java` defines the Hello interface in Java. The file `_example_Hello.java` provides a code framework you can fill in to implement the Hello interface. Following is `Hello.java`.

```
package SendMsg;
public interface Hello extends org.omg.CORBA.Object {
  public java.lang.String sayHello();
}
```

Following is `_example_Hello.java`. To implement the interface, add code to the method marked with the comment **implement operation**.

```
package SendMsg;
public class _example_Hello extends SendMsg._sk_Hello {
  public _example_Hello(java.lang.String name) {
    super(name);
  }
  public _example_Hello() {
    super();
  }
  public java.lang.String sayHello() {
    // implement operation...
    return null;
  }
}
```

After adding code to the example file, save it using a different filename (this sample application uses the name `HelloImpl.java`). Doing this makes clear the difference between the interface and the class that implements it. Also, it preserves your work if you run `idl2java` again and generate another (empty) example file.

Following is `HelloImpl.java`. This file defines the HelloImpl class that implements the Hello interface.

```
// HelloImpl.java
/**
  The HelloImpl class implements the Hello interface
  defined in the file SendMsg.idl.
*/
package SendMsg;

public class HelloImpl extends SendMsg._sk_Hello {
  /** Construct a persistently named object. */
  public HelloImpl(java.lang.String name) {
    super(name);
  }
```

```
  /** Construct a transient object. */
  public HelloImpl() {
    super();
  }

  public String sayHello() {
    // Implement the operation.
    System.out.println("Client called sayHello.");
    return "Hello!";
  }
}
```

Use `javac` to compile the implementation file. The following command compiles `HelloImpl.java`.

```
c:\projects\msg> javac SendMsg\HelloImpl.java
```

Writing a Server Application

The server application (also called an *object server*) creates, activates, and registers services, making them available to clients. In this sample application, the server application is named MsgService. Many of the files used to implement MsgService are contained in the SendMsg package generated by the `idl2java` compiler. The `MsgService.java` file presented here is not generated. Normally you, the programmer, would create this file. This file provides a `main` method that does the following:

- Initializes the Object Request Broker (ORB).
- Initializes the Basic Object Adaptor (BOA). For more information, see Object and Implementation Activation.
- Creates and activates a HelloImpl object.
- Registers the newly created object with the ISB Naming Service.
- Prints out a status message.
- Waits for incoming client requests.

```
// MsgService.java
/**
  This application initializes the BOA, activates the service implementation,
  and registers the service with the Netscape Naming Service.
*/

public class MsgService {
```

```
    public static void main (String[] args) {
        try {
            // Initialize the ORB.
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();
            // The enablePrivilege calls are only required if
            // the service runs on the Communicator. Also, add the
            // Communicator Java files in java40.jar to CLASSPATH.
            /*
            netscape.security.PrivilegeManager.enablePrivilege("UniversalAccept");
            netscape.security.PrivilegeManager.enablePrivilege("UniversalConnect");
            */
            // Initialize the BOA.
            org.omg.CORBA.BOA boa = orb.BOA_init();
            // Activate the object implementation.
            SendMsg.HelloImpl hi = new SendMsg.HelloImpl("IdlExample");
            // Export the newly created object.
            boa.obj_is_ready(hi);

            netscape.WAI.Naming.register("http://myServer.nscp.com/IdlExample", hi);
            System.out.println(hi + " is ready.");
            // Wait for incoming requests.
            boa.impl_is_ready();
        }
        catch(org.omg.CORBA.SystemException e) {
            System.err.println(e);
        }
    }
}
```

The statement that creates and activates a HelloImpl object takes the string "IdlExample" as an argument. The value of this string is arbitrary; however, in this example the same value also appears at the end of the URL passed as a string to the `register` method.

```
netscape.WAI.Naming.register("http://myServer.nscp.com/IdlExample", hi);
```

This string value, plus "NameService" in between the hostname and object, is also used in the client application to resolve a URL and locate the service. For more information about naming and registering objects, see Naming and Binding to Objects.

Use `javac` to compile the server application. The following command compiles `MsgService.java`.

```
c:\projects\msg> javac MsgService.java
```

Writing a Client Application or Applet

The client application (or client applet) locates a service, then invokes methods that service provides. In this sample application, the server application is named MsgClient. Many of the files used to implement MsgClient are contained in the SendMsg package generated by the `idl2java` compiler. The MsgClient.java file presented here is not generated. Normally you, the programmer, would create this file. This client is written to run either from the command line as an application or from within the Communicator or AppletViewer as an applet. Either way, it performs the following functions:

1. Initializes the Object Request Broker (ORB).
2. Locates the Hello service.
3. Calls the `sayHello` method provided by the Hello service.
4. Prints the resulting message.

Listing 2.2 MsgClient.java provides the client-side program, written in Java.

```
// MsgClient.java
/**
  You can run this client code as an application from the command line
  or as an applet from within a Java-enabled browser. It initializes the
  ORB, locates the service, and calls a service method to print a message.
*/
import netscape.WAI.Naming;

public class MsgClient extends java.applet.Applet {
    String _url = "http://myServer.nscp.com/NameService/IdlExample";
    public void init() {
        try {
            // Initialize the ORB.
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();

            // Locate a message service.
            org.omg.CORBA.Object obj = Naming.resolve(_url);
            SendMsg.Hello hi = SendMsg.HelloHelper.narrow(obj);

            // Print a message.
            System.out.println(hi.sayHello());
        }
        catch(org.omg.CORBA.SystemException e) {
            System.err.println(e);
        }
    }
```

```
    public static void main(String args[]) {
        MsgClient mc = new MsgClient();
        mc._url = "http://myServer.nscp.com/IdlExample";
        mc.init();
    }
}
```

Use `javac` to compile the client application. The following command compiles `MsgClient.java`.

```
c:\projects\msg> javac MsgService.java
```

Starting a Server Application

To start a server application, open a console window and use the `java` command with the -DDISABLE_ORB_LOCATOR flag. For example, the following command starts the MsgService application.

```
c:\projects\msg> java -DDISABLE_ORB_LOCATOR MsgService
```

If the application starts successfully, it will print a message like the following in the console window.

```
SendMsg.HelloImpl[Server,oid=PersistentId[repId=IDL:SendMsg/
Hello:1.0,objectName=IdlExample]] is ready
```

If there is a problem, you may see a message like the following.

```
C:\CAFE\PROJECTS\IDLBANK>java -DDISABLE_ORB_LOCATOR MsgService
org.omg.CORBA.INV_OBJREF[completed=MAYBE, reason=Invalid object ref returned by URL
resolver]
```

This message is typically caused by a statement that calls `register`, for example:

```
netscape.WAI.Naming.register("http://myServer.nscp.com/isbBank/AcctMgr", manager);
```

where "myServer.nscp.com" is the name of the host and "isbBank" is the name of a document directory known to the Enterprise Server. The Enterprise Server must be configured to enable Web Publishing, and the specified document directory must be configured to allow write access. If you have other problems starting the application, refer to Troubleshooting.

Starting a Client Application

After starting the server application, open a different console window and use the `java` command with the -DDISABLE_ORB_LOCATOR flag. For example, the following command starts the MsgClient application.

```
c:\projects\msg> java -DDISABLE_ORB_LOCATOR MsgClient
```

If the application starts successfully, it will print the following message in the console window.

```
Hello!
```

If your development environment is not connected to a network (that is, if you are developing the application on a stand-alone system), it may take several seconds for the message to appear. Response times are generally faster on networked systems. If you have other problems starting the application, refer to Troubleshooting.

Advanced Networking Options

You can change the network address and port used by a client or server application by setting the `OAPort` property when you start the application. If a port number is not specified, an unused port will be selected. For example, the following command starts the MsgService application and specifies port 1234.

```
c:\projects\msg> java -DDISABLE_ORB_LOCATOR -DOAPort=1234 MsgService
```

Running a Client Applet

You can run a client applet in an HTML page using either the AppletViewer or the Communicator. The HTML page must include an APPLET tag to load the client class. For example, the following page loads the MsgClient class.

```
<HTML>
<BODY>
This applet prints a message in the Java Console.
<BR>
<APPLET CODE=MsgClient.class WIDTH=200 HEIGHT=100>
</applet>
</body>
</html>
```

Communicator

Choose File - Open Page, then enter the URL for the HTML page that runs the client.

```
http://myServer.nscp.com/projects/msg/MsgClient.html
```

If the applet starts successfully, it will print the following message in the Java Console window.

```
Hello!
```

Some problems running applets in the Communicator are caused by CLASSPATH settings. It may help to empty the CLASSPATH before starting Communicator. To empty the CLASSPATH, enter the following command:

```
c:\> set CLASSPATH=
```

If you have other problems starting the applet, refer to Troubleshooting.

AppletViewer

Before testing client applets in the AppletViewer, add the following lines to the `appletviewer.properties` file (by default, it's in the `\lib` subdirectory of the JDK distribution):

```
DISABLE_ORB_LOCATOR=1
browser.vendor.applet=1
```

To run a client applet in the AppletViewer, enter the filename of an HTML file that loads the client class. For example, the following command loads MsgClient.

```
c:\projects\msg> appletviewer MsgClient.html
```

Chapter

3

Naming and Binding to Objects

This chapter describes techniques for naming and operating on objects, object references, and interfaces. It includes the following major sections:

- Web Naming Service
- Operations on Object References
- Interface and Object Names

Web Naming Service

The Web Naming Service enables you to associate a *Uniform Resource Locator* (URL) with an object. Once a URL has been bound to an object, clients can obtain a reference to the object by specifying the URL as a string instead of the object's name. This feature enables client applications to locate objects provided by any vendor.

The key methods are provided by the `netscape.WAI.Naming` class. This class is part of the `nisb.zip` file in the Netscape Enterprise Server and the `iiop10.jar` file in Netscape Communicator. The methods behaves slightly differently depending on which ORB is used. Differences are described below.

register

This method uses WebNaming to register an object with a URL of the form `http://host[:port]/path/name`. The caller's machine must be allowed to do an HTTP 'PUT' on the designated host.

Calling **register** using the EnterpriseServer's ORB is different than calling **register** using the Communicator's ORB. With the EnterpriseServer, this call automatically prepends `NameService/` to the specified `path/name`. For example:

```
register("http://mikelee/HelloService", service);
```

registers the object with the following (automatically modified) URL:

```
http://mikelee/NameService/HelloService
```

With the Communicator, the URL is not automatically modified, so you need to specify the `NameService/` prefix yourself. For example:

```
register("http://mikelee/NameService/HelloService", service);
```

If the object has already been registered with a URL, this method replaces (overwrites) the old URL with a new one.

resolve

This method resolves a CORBA object from a URL of the form `http://host[:port]/path/name`.

Calling **resolve** using the EnterpriseServer's ORB is different than calling **resolve** using the Communicator's ORB. With the EnterpriseServer, this call automatically prepends `NameService/` to the specified `path/name`. For example:

```
obj = resolve("http://mikelee/HelloService");
```

resolves a CORBA object using the following (automatically modified) URL:

```
http://mikelee/NameService/HelloService
```

With the Communicator, the URL is not automatically modified, so you need to specify the `NameService/` prefix yourself. For example:

```
obj = resolve("http://mikelee/NameService/HelloService");
```

The following code examples show how these methods are used by clients and services. The first example is part of a service application that calls `register` to register a service with the Enterprise Server's ORB.

```
// Part of a service application using the Enterprise Server's ORB.
public class HelloService extends _sk_HelloWorld {
   public static void main(String args[]) {
      // Initialize the BOA because we are going to accept connections.
      BOA boa = orb.BOA_init();
      /* Create the service. */
      HelloService service = new HelloService();
      /* Expose the service to the net. */
      boa.obj_is_ready(service);
      /* Register the service. */
      try {
         String host =
            java.net.InetAddress.getLocalHost().getHostName();
         Naming.register("http://" + host + "/HelloService", service);

         // For the Communicator's ORB this call would be:
         // Naming.register("http://" + host +
         //                 "/NameService/HelloService", service);

      } catch (Exception e)
      { e.printStackTrace();
        System.exit( -1 );
      }
      /* Wait for requests. */
      boa.impl_is_ready();
   }
}
```

Following is part of a client applet that calls resolve to find a service, given a URL. Applets always use the Communicator's ORB.

```
// Part of a client applet using the Communicator's ORB.
public class HelloClientApplet extends java.applet.Applet {
   org.omg.CORBA.ORB orb;
   HelloWorld hello;
   public void init() {
      /* Initialize the ORB. */
      orb = org.omg.CORBA.ORB.init();
      /* Resolve an object. */
      // String host = getCodeBase().getHost();
      String host = "myServer.nscp.com";
      // From applets you can only use the Communicator's ORB.
      String url_string = "http://" + host +
         "/NameService/HelloService";
      org.omg.CORBA.Object obj = Naming.resolve(url_string);
      /* In some cases (where multiple CORBA interfaces
         are implemented), casting requires a query to the
         remote host. So we call 'narrow' to cast here. */
      hello = HelloWorldHelper.narrow(obj);
      if(hello == null)
```

```
            unableToLocate(url_string); // bad cast returns null
        setLayout(new GridLayout(1,1));
        Button b = new Button("GO!");
        add(b);
    }
}
```

Operations on Object References

Given an object reference, a client can invoke methods defined in that object's IDL specification. In addition, all objects derived from the class `org.omg.CORBA.Object` inherit methods that you can use to manipulate the object. Some of these methods are listed below; for a complete discussion of these and other methods, see the *Netscape Internet Service Broker for Java Reference Guide*.

Note You cannot use the `instanceof` keyword to determine the runtime type.

Table 3.1 From org.omg.CORBA.Object, methods on object references

Method	Description
_clone	Creates a copy of the object. It also creates another TCP/IP connection.
_is_a	Determines whether an object implements a specified interface.
_is_bound	Returns true if a connection is currently active for this object.
_is_equivalent	Returns true if two objects refer to the same interface implementation.
_is_local	Returns true if this object is implemented in the local address space.
_is_remote	Returns true if this object's implementation resides on a remote host.
_object_name	Returns this object's name.

Converting an Object Reference to a String

Object references are opaque and can vary from one ORB to another, so ISB for Java provides methods that allow you to convert an object reference to a string as well as convert a string back into an object reference. The CORBA specification refers to this process as "stringification."

Method	Description
object_to_string	Converts an object reference to a string.
string_to_object	Converts a string back to an object reference.

Narrowing Object References

The process of converting an object reference's type from a general super-type to a more specific sub-type is called *narrowing*.

Note You cannot use the Java casting facilities for narrowing.

ISB for Java maintains a type graph for each object interface so that narrowing can be accomplished by using the object's narrow method. If the `narrow` method determines it is not possible to narrow an object to the type you request, it will return NULL.

Listing 3.1 The narrow method generated for the AccountManager.

```
abstract public class AccountManagerHelper {
  ...
  public static Bank.AccountManager narrow(org.omg.CORBA.Object object)
{
      ...
  }
  ...
}
```

Interface and Object Names

Interface names and object names enable a client to use multiple instances of a service.

Interface Names

When you define an object's interface in an IDL specification, you give it an *interface name*. For example, the following IDL code specifies interfaces named Account and AccountManager.

```
module Bank {
    interface Account {
```

```
        float balance();
    };
    interface AccountManager {
        Account open(in string name);
    };
};
```

Object Names

When creating an object, a service application must specify an *object name* to make it available to clients. When the service calls `BOA.object_is_ready`, the service's interface name is registered only if the object is named. Objects that are given an object name when they are created return *persistent object references*. See Object and ImplementationActivation for a complete discussion of persistent and transient objects. To make a client use a specific service, provide an object name. Also, a client must specify object names to bind to more than one instance of a service at a time. The object name distinguishes between multiple instances of a service. If an object name is not specified, the ORB will return any suitable object with the specified interface.

Using Qualified Object Names with Servers

Consider a banking application where you need to have two AccountManager objects available; one for a bank in Chicago and one for another bank in Dallas. You may even want to implement two separate object servers, possibly on different hosts. Each server could instantiate its own AccountManager object, but each would use the AccountManager constructor that accepts an object name. The following code examples show the server code for creating AccountManagers named Dallas and Chicago, and the client code to connect to the Dallas AccountManager.

Listing 3.2 Creating AccountManagers with the object names Dallas and Chicago.

```
// Using Enterprise Server's ORB
public class Server {
  public static void main(String[] args) {
    try {
      // Initialize the ORB.
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();
       // Initialize the BOA.
      org.omg.CORBA.BOA boa = orb.BOA_init();
      // Create the account manager objects.
      AccountManager dalMngr= new AccountManager("Dallas");
      AccountManager chiMngr= new AccountManager("Chicago");
      // Export the newly created objects.
      boa.obj_is_ready(dalMngr);
```

```
      boa.obj_is_ready(chiMngr);

      // Or, if using Communicator's ORB, the call would be:
      // netscape.WAI.Naming.register
      //    ("http://myServer.nscp.com/NameService/Dallas", dalMngr);

      netscape.WAI.Naming.register
         ("http://myServer.nscp.com/Dallas", dalMngr);
      netscape.WAI.Naming.register
         ("http://myServer.nscp.com/Chicago", chiMngr);

      // Optional status messages.
      System.out.println(dalMngr + " is ready.");
      System.out.println(chiMngr + " is ready.”);
      // Wait for incoming requests.
      boa.impl_is_ready();
    } catch (org.omg.CORBA.SystemException se) {
        System.err.println(se);
    }
  }
}
```

Listing 3.3 A client binding to an AccountManager with the object name Dallas.

```
// Using Enterprise Server's ORB
public class Client {
  public static void main(String args[]) {
    try {
      // Initialize the ORB.
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();
      // Locate an account manager named Dallas.
      // Or, if using Communicator's ORB, the urlStr should be:
      //    "http://myServer.nscp.com/NameService/Dallas";
      String urlStr = "http://myServer.nscp.com/Dallas");
      org.omg.CORBA.Object obj = netscape.WAI.Naming.resolve(urlStr);
      Bank.AccountManager manager =
         Bank.AccountManagerHelper.narrow(obj);
      // Use args[0] as the account name, or a default.
      String name = args.length > 0 ? args[0] : "Jack B. Quick";
      // Request the account manager to open a named account.
      Bank.Account account = manager.open(name);
      // Get the balance of the account.
      float balance = account.balance();
      // Print out the balance.
      System.out.println
        ("The balance in " + name + "'s account is $" + balance);
    } catch (org.omg.CORBA.SystemException(se) {
        System.err.println(se);
    }
  }
}
```

Chapter

4

Object and Implementation Activation

This chapter discusses how services are implemented and made available to clients. It includes the following major sections:

- Object Implementation
- Activating Objects Directly
- The Basic Object Adaptor
- Object and Implementation Deactivation

Object Implementation

An object implementation provides the state and processing activities for an ORB object. An ORB object is created when its implementation class is instantiated in Java by an implementation processor service. An object implementation uses the Basic Object Adaptor (BOA) to activate its ORB objects for use by clients. ISB for Java supports both *persistent* and *transient* object references.

Persistent Object References

You create a *persistent object reference* when you instantiate an object and specify an object name. Persistent object references remain valid beyond the lifetime of the processes that create them. These object references have a global scope; they are registered with the naming service and used by client applications. Persistent object references are registered when the `boa.obj_is_ready` method is invoked.

You can use persistent object references to implement long-running services that provide long-term tasks. Light-weight and domain-specific, transient object references take a relatively short amount of time to instantiate. Persistent object references take longer to instantiate. Consequently, the ideal mix is to have some persistent object references and many more transient object references used by your application. The following code example creates and registers a persistent object reference using the name ISB_Bank.

Listing 4.1 Creating a persistent object.

```
// Using Enterprise Server's ORB
public class Server {
  public static void main(String[] args) {
    try {
      // Initialize the ORB.
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();
      // Initialize the BOA.
      org.omg.CORBA.BOA boa = orb.BOA_init();
      // Create the account manager object.
      AccountManager manager = new AccountManager("ISB_Bank");
      // Export the newly created object.
      boa.obj_is_ready(manager);
      String host = java.netInetAddress.getLocalHost().getHostName();
      netscape.WAI.Naming.register
         ("http://" + host + "/ISB_Bank", manager);

      // Or, if using Communicator's ORB, the call would be:
      // netscape.WAI.Naming.register
      //    ("http://" + host + "/NameService/ISB_Bank", manager);

      // Wait for incoming requests
      boa.impl_is_ready();
    } catch(org.omg.CORBA.SystemException se) {
        System.err.println(se);
    }
      catch(java.net.UnknownHostException u) {
        System.err.println(u);
    }
  }
}
```

Checking for Persistent Object References

The `Object._is_persistent` method allows your client application to determine whether an object reference is persistent or transient. It is important to know whether a reference is persistent because some methods for manipulating object references will fail if the reference is transient. The `_is_persistent` method returns true if the reference is persistent and false if it is transient. For information about all of the available methods, see the *Netscape Internet Service Broker for Java Reference Guide*.

Transient Object References

Object references that are only available during the lifetime of the process that created them are called *transient object references*. Only those entities that possess an explicit object reference to a transient object reference can invoke its methods. To create a transient object reference, instantiate an object without specifying an object name, as shown below.

Listing 4.2 Creating a transient object.

```
// Using Enterprise Server's ORB
public class Server {
  public static void main(String[] args) {
    try {
      // Initialize the ORB.
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();
      // Initialize the BOA.
      org.omg.CORBA.BOA boa = orb.BOA_init();
      // Create the account manager object.
      AccountManager manager = new AccountManager();
      // Export the newly created object.
      boa.obj_is_ready(manager);
      String host = java.netInetAddress.getLocalHost().getHostName();
      netscape.WAI.Naming.register
         ("http://" + host + "/ISB_Bank", manager);

      // Or, if using Communicator's ORB, the call would be:
      // netscape.WAI.Naming.register
      //    ("http://" + host + "/NameService/ISB_Bank", manager);

      // Wait for incoming requests
      boa.impl_is_ready();
    } catch(org.omg.CORBA.SystemException se) {
        System.err.println(se);
    }
```

```
      catch(java.net.UnknownHostException u) {
        System.err.println(u);
    }
  }
}
```

Object Registration

Once a server has instantiated the ORB objects that it offers, the BOA must be notified when the objects have been initialized. Lastly, the BOA is notified when the server is ready to receive requests from client applications.

The `obj_is_ready` method notifies the BOA that a particular ORB object is ready to receive requests from client applications. If your server offers more than one ORB object, it must invoke `obj_is_ready` for each object, passing the object reference as an argument.

If a persistent object reference is passed to `obj_is_ready`, the BOA will register the object with the naming service. If the reference is transient, no such registration will occur. If `obj_is_ready` has not been invoked for an object by its server and a client attempts to bind to the object, the exception NO_IMPLEMENT is raised.

Once all of the objects have been instantiated and all the invocations of `obj_is_ready` have been made, the server must invoke the `impl_is_ready` method to enter an event loop and await client requests. Servers that are activated through a GUI (graphical user interface) or are activated by other means do not need to use `impl_is_ready`.

Activating Objects Directly

Direct activation of an object involves an object server instantiating all the Java implementation classes, invoking the `BOA.obj_is_ready` method for each object, and then invoking `BOA.impl_is_ready` to begin receiving requests. The following code example shows how this processing would occur for a server offering two AccountManager objects; one with the object name of Chicago and the other named Dallas. Once the objects have been instantiated and activated, the server invokes `BOA.impl_is_ready` to begin receiving client requests.

Note The method `BOA.obj_is_ready` must be called for each object offered by the implementation.

Listing 4.3 Server activating two objects and the implementation.

```
// Using Enterprise Server's ORB
public class Server {
  public static void main(String[] args) {
    try {
      // Initialize the ORB.
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();
      // Initialize the BOA.
      org.omg.CORBA.BOA boa = orb.BOA_init();
      // Create the account manager object for Chicago.
      AccountManager chicago = new AccountManager("Chicago");
      // Create the account manager object for Dallas.
      AccountManager dallas = new AccountManager("Dallas");
      // Export the newly created objects.
      boa.obj_is_ready(chicago);
      boa.obj_is_ready(dallas);
      System.out.println(chicago + " is ready.");
      System.out.println(dallas + " is ready.");
      String host = java.netInetAddress.getLocalHost().getHostName();

      // Or, if using Communicator's ORB, the call would be:
      // netscape.WAI.Naming.register
      //    ("http://" + host + "/NameService/bank", chicago);

      netscape.WAI.Naming.register("http://" + host + "/bank", chicago);
      netscape.WAI.Naming.register("http://" + host + "/bank", dallas);

      // Wait for incoming requests
      boa.impl_is_ready();
    } catch(org.omg.CORBA.SystemException se) {
        System.err.println(se);
    }
      catch(java.net.UnknownHostException u) {
        System.err.println(u);
    }
  }
}
```

The Basic Object Adaptor

A service implementation uses the Basic Object Adaptor (BOA) to activate its ORB objects so they can be used by client applets and applications. ISB for Java's BOA providesseveralimportantfunctionstoclientsandtheserviceimplementationstheyuse.

A service can reside in the same process as its client or it can reside in a separate process called a *server*. Servers can contain and offer one or more services. Furthermore, they can be activated by the BOA on demand or they can be started by some entity external to the BOA.

Object and Implementation Deactivation

An object implementation that was started manually is deactivated when the server that implements the object exits. At that time, ISB for Java automatically unregisters the services within that implementation.

Service implementations that called `obj_is_ready` can be deactivated explicitly by calling `deactivate_obj`. Then, the implementation will not be available to service client requests. The service can only be re-activated if it is restarted or if it again calls `obj_is_ready`.

Chapter

5

The Dynamic Invocation Interface

This chapter describes how clients can dynamically create requests for services at run time. It includes the following major sections:

- Overview
- Obtaining an Object Reference
- Creating and Initializing a Request
- Sending a DII Request

Overview

The *Dynamic Invocation Interface* (DII) enables clients to invoke operations on any registered service without having to first compile the client stubs that were created for that service by the IDL compiler. By using DII, a client can dynamically build operation requests for any interface that has been stored in the Interface Repository. Service implementations do not require any special design to be able to receive and handle DII requests.

DII is not as efficient as static operation requests, but it offers some important advantages. Clients are not restricted to using the service that were defined at the time the client was compiled. In addition, clients that use DII do not need to be recompiled to access newly-activated service implementations.

Steps for Dynamic Invocation

These are the steps that a client follows when using DII.

1. Obtain an object reference to the service you wish to use.
2. Create a Request object for the service.
3. Initialize the request parameters and the result to be returned.
4. Invoke the request and wait for the results.
5. Retrieve the results.

Using the idl2java Compiler

The `idl2java` compiler has a flag (`-portable`) which, when switched on, generates stub code using DII. To understand how to do any type of DII, create an IDL file, generate with `-portable`, then look at the stub code.

Obtaining an Object Reference

Examples in this chapter obtain object references by using the `register` and `resolve` methods provided by ISB for Java's Web Naming Service. See also Operations on Object References.

Creating and Initializing a Request

When a client invokes a method on a service, a Request object is created to represent the method invocation. The Request object is written, or *marshalled*, to a buffer and sent to the service implementation. When a client uses client stubs, this processing occurs transparently. Clients that use DII must create and send the Request object themselves.

There is no constructor for Request objects. Use the `Object._request` method or one of the `Object._create_request` methods to create a Request object. The following code shows the methods offered by the Object interface. For information about all of the available methods, see the *Netscape Internet Service Broker for Java Reference Guide*.

Listing 5.1 The Object methods for creating a Request object.

```
package org.omg.CORBA;

    public interface Object {
        ...
        public org.omg.CORBA.Request _request(
            java.lang.String operation
        );

        public org.omg.CORBA.Request _create_request(
            org.omg.CORBA.Context ctx,
            java.lang.String operation,
            org.omg.CORBA.NVList arg_list,
            org.omg.CORBA.NamedValue result
        );

        public org.omg.CORBA.Request _create_request(
            org.omg.CORBA.Context ctx,
            java.lang.String operation,
            org.omg.CORBA.NVList arg_list,
            org.omg.CORBA.NamedValue result,
            org.omg.CORBA.TypeCode[] exceptions,
            java.lang.String[] contexts
        );
        ...
}
```

The _create_request Methods

You can use the `_create_request` method to create a Request object, initialize the Context, the operation name, the argument list to be passed and the result. The `request` parameter points to the Request object that was created for this operation. The second form of this method allows you to specify a list of Context objects for the request; this supports type checking on context names.

The _request Method

The following code example shows the use of the `_request` method to create a Requestobject,specifyingonlytheoperationname.OnceaRequestobjectiscreated,the arguments, if any, must be set before the request can be sent to the server.

Listing 5.2 A portion of the Client.java file, showing the creation of a Request object.

```
// Using Enterprise Server's ORB
// Client.java
import org.omg.CORBA.*;
public class Client {
  public static void main(String[] args) {
    org.omg.CORBA.Object manager, account;
    try {
      orb = org.omg.CORBA.ORB.init();
      // Locate an account manager.
      _manager = Bank.AccountManagerHelper.narrow
                  (netscape.WAI.Naming.resolve("http://myHost/bank"));
      // Or, if using Communicator's ORB, the call would be:
      // netscape.WAI.Naming.resolve("http://myHost/NameService/bank");
    }
    catch(Exception e) {
      e.printStackTrace();
    }
    String name = args.length > 0 ? args[0] : "Jack B. Quick";
    {
      Request openReq = manager._request("open");
      openReq.add_in_arg().insert_string(name);
      openReq.set_return_type(orb.get_primitive_tc(TCKind.tk_objref));
      openReq.invoke();
      account = openReq.result().value().extract_Object();
    }
    {
      Request balanceReq = account._request("balance");
      balanceReq.set_return_type
        (orb.get_primitive_tc(TCKind.tk_float));
      balanceReq.invoke();
      float balance = balanceReq.result().value().extract_float();
      System.out.println
        ("The balance in " + name + "'s account is $" + balance);
    }
  }
}
```

Setting the Arguments

The arguments for a Request are represented with a `NVList` object, which stores name-value pairs as `NamedValue` objects. You can use the `Request.arguments` method to obtain a reference to the argument list. This reference can then be used to set the names and values of each of the arguments.

Optionally, the arguments for a `Request` can be set using the various `add_` methods. In the previous code example, the `add_in_arg` method is used to set the argument for the `open` method on the `AccountManager` interface.

NVList

`NVList` provides a list of `NamedValue` objects that represent the arguments for a method invocation. There is no constructor. Use the `ORB.create_list` method or the `ORB.create_operation` method to create an `NVList` object.

`NVList` provides methods for adding, removing, and querying the objects in the list. A complete description of `NVList` can be found in the *Netscape Internet Service Broker for Java Reference Guide*.

Listing 5.3 The NVList interface.

```
package org.omg.CORBA;
public interface NVList {
  public int count();
  public void add(int flags);
  public void add_item(java.lang.String name, int flags);
  public void add_value(java.lang.String name,
                        org.omg.CORBA.Any value,
                        int flags);
  public org.omg.CORBA.NamedValue item(int index);
  public void remove(int index);
}
```

NamedValue

`NamedValue` holds a name-value pair that can be used to represent both input and output arguments for a method invocation request. It is also used to represent the result of a request that is returned to the client application. There is no constructor for `NamedValue`. Use the `ORB.create_named_value` method to create a `NamedValue` object.

The `name` property is a character string and the `value` property is represented by an `Any` class. A complete description of `NamedValue` can be found in the *Netscape Internet Service Broker for Java Reference Guide*.

Listing 5.4 The NamedValue interface.

```
package org.omg.CORBA;

public interface NamedValue {
    public java.lang.String name();
    public org.omg.CORBA.Any value();
    public int flags();
}
```

Table 5.1 The NamedValue methods.

Method	Description
name	Returns a reference to the name of the item that you can then use to initialize the name.
value	Returns a reference to an Any object representing the item's value that you can then use to initialize the value. For more information, see The Any Class.

flags	Indicates if this item is an input argument, an output argument or both an input and output argument. If the item is both an input and output argument, you can specify a flag indicating that the ORB should make a copy of the argument and leave the caller's memory intact. Flags are:
	`ARG_IN`
	`ARG_OUT`
	`ARG_INOUT`

The Any Class

The `Any` class holds an IDL type so that it can be passed in a type-safe manner. Objects of this class have a reference to a TypeCode that defines the contained object's type and a reference to the contained object. There is no constructor for this class. The `ORB.create_any` method creates an `Any` object.

Methods are provided to create, read, write, and test the equality of Any objects as well as initialize and query the object's value and type. A complete description can be found in the *Netscape Internet Service Broker for Java Reference Guide*.

Listing 5.5 The Any class.

```
package org.omg.CORBA;

abstract public class Any {
    public static Any create();
    abstract public TypeCode type();
    abstract public void type(TypeCode type);
    abstract public void read_value(InputStream input, TypeCode type);
    abstract public void write_value(OutputStream output);
    abstract public boolean equal(Any rhs);
    ...
}
```

The TypeCode Class

This class is used by the Interface Repository and the IDL compiler to represent the type of arguments or attributes. `TypeCode` objects are also used in a `Request` to specify an argument's type, in conjunction with the `Any` class. There is no constructor for this class. Use the `ORB.get_primitive_tc` method or one of the `ORB.create_*_tc` methods to create a `TypeCode` object.

`TypeCode` objects have a `kind` property, represented by one of the values defined by the `TCKind` class. Complete descriptions of these classes can be found in the *Netscape Internet Service Broker for Java Reference Guide*.

Listing 5.6 The TypeCode Class.

```
abstract public class TypeCode {

  abstract public boolean equal(org.omg.CORBA.TypeCode tc);
  abstract public org.omg.CORBA.TCKind kind();
  abstract public java.lang.String id()
            throws org.omg.CORBA.TypeCodePackage.BadKind;
  abstract public java.lang.String name()
            throws org.omg.CORBA.TypeCodePackage.BadKind;
  abstract public int member_count()
            throws org.omg.CORBA.TypeCodePackage.BadKind;
  abstract public java.lang.String member_name(int index)
            throws org.omg.CORBA.TypeCodePackage.BadKind,
                   org.omg.CORBA.TypeCodePackage.Bounds;
  abstract public org.omg.CORBA.TypeCode member_type(int index)
            throws org.omg.CORBA.TypeCodePackage.BadKind,
                   org.omg.CORBA.TypeCodePackage.Bounds;
  abstract public org.omg.CORBA.Any member_label(int index)
            throws org.omg.CORBA.TypeCodePackage.BadKind,
                   org.omg.CORBA.TypeCodePackage.Bounds;
  abstract public org.omg.CORBA.TypeCode discriminator_type()
```

```
            throws org.omg.CORBA.TypeCodePackage.BadKind;
  abstract public int default_index()
            throws org.omg.CORBA.TypeCodePackage.BadKind;
  abstract public int length()
            throws org.omg.CORBA.TypeCodePackage.BadKind;
  abstract public org.omg.CORBA.TypeCode content_type()
            throws org.omg.CORBA.TypeCodePackage.BadKind;
}
```

Sending a DII Request

The Request interface provides several methods for sending a request, once it has been properly initialized. The simplest of these is the `invoke` method, which sends the request and waits for a response before returning to your client application.

Receiving the Result

The `result` method returns a reference to a `NamedValue` object that represents the return value.

Listing 5.7 A portion of the Client.java file, showing how a request is sent to the object implementation.

```
...
{
   Request openReq = manager._request("open");
   openReq.add_in_arg().insert_string(name);
   openReq.set_return_type(orb.get_primitive_tc(TCKind.tk_objref));
   openReq.invoke();
   account = openReq.result().value().extract_Object();
}
{
   Request balanceReq = account._request("balance");
   balanceReq.set_return_type(orb.get_primitive_tc(TCKind.tk_float));
   balanceReq.invoke();
   float balance = balanceReq.result().value().extract_float();
   System.out.println
        ("The balance in " + name + "'s account is $" + balance);
}
...
```

The send_deferred Method

A non-blocking method, `send_deferred`, is also provided for sending operation requests. It allows your client to send the request and then use the poll_response method to determine when the response is available. The `get_response` method blocks until a response is received. The following code examples show how these methods are used.

Listing 5.8 A portion of the account_clnt2.java file showing the use of the send_deferred and poll_response methods.

```
...
// Send the request
try {
    req.send_deferred();
    System.out.println("Send deferred call is made...");
} catch(org.omg.CORBA.SystemException e) {
    System.out.println("Error while sending request");
    System.err.println(e);
}

// Poll for response
try {
    while (!req.poll_response())
    {
        try{
            System.out.println("Waiting for response");
            Thread.sleep(1000);
        } catch (Exception e){
            System.err.println(e);
        }
    }
} catch (org.omg.CORBA.SystemException e) {
    System.out.println("Failure while polling for response");
    System.err.println(e);
}

try {
    req.get_response();
    // Get the return value;
    balance = req.result().value().extract_float();
} catch(org.omg.CORBA.SystemException e) {
    System.out.println("Error while receiving response");
    System.err.println(e);
}
...
```

The send_oneway Method

The `send_oneway` method can be used to send an asynchronous request. Oneway requests do not involve a response being sent from the object implementation.

Sending Multiple Requests

If you create a sequence of DII Request objects, the entire sequence can be sent using the ORB methods `send_multiple_requests_oneway` or `send_multiple_requests_deferred`. If the sequence of requests is sent as oneway requests, no response is expected from the server to any of the requests.

Receiving Multiple Requests

When a sequence of requests is sent using `send_multiple_requests_deferred`, the `poll_next_response` and `get_next_response` methods are used to receive the responses the server sends for each request.

The ORB method `poll_next_response` can be used to determine if a response has been received from the server. This method returns 1 if there is at least one response available. This method returns zero if there are no responses available.

The ORB method `get_next_response` can be used to receive a response. If no response is available, this method will block until a response is received. If you do not wish your client application to block, use the `poll_next_response` method to first determine when a response is available and then use the `get_next_response` method to receive the result.

Listing 5.9 ORB methods for sending multiple requests and receiving the results.

```
package org.omg.CORBA;

    abstract public class ORB {
        abstract public org.omg.CORBA.Environment create_environment();
        abstract public void send_multiple_requests_oneway
            (org.omg.CORBA.Request[] reqs);
        abstract public void send_multiple_requests_deferred
            (org.omg.CORBA.Request[] reqs);
        abstract public boolean poll_next_response();
        abstract public org.omg.CORBA.Request get_next_response();
```

```
            ...
}
```

Chapter

6

The Tie Mechanism

This chapter describes how to use the tie mechanism in place of inheritance. The tie mechanism offers an alternative when it is not convenient or possible to have your implementation class inherit from the ISB for Java skeleton class. This chapter includes the following major sections:

- Overview
- The Tie Example Program

Overview

The tie mechanism enables an object to support `org.omg.CORBA.Object` *and* inherit from another class. You can use it to work around Java's single-inheritance restriction when it is not convenient or possible to have your implementation class inherit from the ISB for Java skeleton class.

The tie mechanism provides a *delegator implementation* class that inherits from `org.omg.CORBA.Object`. The delegator implementation does not provide any semantics of its own. Instead, it delegates every call to the real implementation class, which can be implemented separately and can inherit from any class it wishes.

Inheritanceiseasiertousethanthetiemechanismbecauseimplementationobjectslook and behave just like object references. If a client uses an implementation object in the same process, method calls involve less overhead because no transport, indirection, or delegation of any kind is required.

The Tie Example Program

Thetieexampleisverysimilartothe*sampleapplication*presentedin"GettingStarted"and assumes you are familiar with that application. The key differences are:

- Generating Tie Files
- Implementing Tie Interfaces
- Writing a Tie Server Application

The procedures for developing and running a client application or applet are the same whether you use the tie mechanism or not. This sample application was developed under Windows NT 4.0. It assumes that `c:\projects\tieMsg` is its root directory, that some files reside there, some in a subdirectory named `SendMsg`. If you recreate this sample application using a different configuration, change code and commands accordingly.

Generating Tie Files

Following is the IDL file (named `SendMsg.idl`) for this example.

```
module SendMsg {
    interface Hello {
        string sayHello();
    };
};
```

The `idl2java` compiler generates files to support the tie mechanism by default. The following command compiles `SendMsg.idl` (the `-no_comments` flag suppresses comments in generated files, reducing clutter for this example).

```
c:\projects\tieMsg> java2idl -no_comments SendMsg.idl
```

Implementing Tie Interfaces

The `idl2java` compiler generates a file containing a code framework you can use to implement interfaces (in this example, the file is `_example_Hello.java`). After adding your code, save the file using a different name (this example uses the name `HelloImpl.java`). `idl2java` also generates an Operations file for each interface in a module. The Operations file provides the Java definition of the interface. To use the tie mechanism, a class implements the corresponding Operations interface. Here, HelloImpl implements HelloOperations. This class is also free to extend another class, if desired; without the tie mechanism, this class would have to extend the skeleton class SendMsg._sk_Hello.

```
// HelloImpl.java
/**
  The HelloImpl class implements the HelloOperations interface
  defined in the file HelloOperations.java generated by idl2java.
*/
package SendMsg;
/*
   With the tie mechanism, this class can extend
   a class other than SendMsg._sk_Hello.
   Without the tie mechanism:
   public class HelloImpl extends SendMsg._sk_Hello {
*/
public class HelloImpl implements SendMsg.HelloOperations {
/* This constructor is included when idl2java generates _example_Hello.java,
   but it is not valid in this example.
   public HelloImpl(java.lang.String name) {
     super(name);
  }
*/

  public HelloImpl() {
    super();
  }

  public String sayHello() {
    // Implement the operation.
    System.out.println("Tie Client called sayHello.");
    return "Tie Hello!";
  }
}
```

Writing a Tie Server Application

The code for a server application is largely the same whether you use the tie mechanism or not. When using the tie mechanism, the key step is to initialize a _tie class with an instance of an implementation class to which it delegates every operation. The following example instantiates the implementation class HelloImpl, then passes that instance to the constructor for the _tie class named _tie_Hello.

```
// Using Enterprise Server's ORB
// MsgService.java
public class MsgService {
   public static void main (String[] args) {
      try {
         // Initialize the ORB.
         org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();
         // Initialize the BOA.
         org.omg.CORBA.BOA boa = orb.BOA_init();

         // Activate the object implementation.
         // Without tie mechanism:
         // SendMsg.HelloImpl hi = new SendMsg.HelloImpl("TieExample");
         SendMsg.HelloImpl dlgt = new SendMsg.HelloImpl();
         SendMsg.Hello hi = new SendMsg._tie_Hello(dlgt, "TieExample");

         // Export the newly created object.
         boa.obj_is_ready(hi);
         netscape.WAI.Naming.register("http://myHost/TieExample", hi);

         // Or, if using Communicator's ORB, the call would be:
         // netscape.WAI.Naming.register("http://myHost/NameService/TieExample", hi);

         System.out.println(hi + " is ready.");
         // Wait for incoming requests.
         boa.impl_is_ready();
      }
      catch(org.omg.CORBA.SystemException e) {
         System.err.println(e);
      }
   }
}
```

The `_tie_Hello.java` file (shown below) is generated by the `idl2java` compiler. It implements methods by calling the corresponding method exposed by the member _delegate, an instance of a class that implements the operations defined in HelloOperations. For example, when a client calls the `sayHello` method, the

_tie_Hello class calls the `sayHello` method provided by _delegate. In the server application (MsgService.java), _delegate is initialized with an instance of HelloImpl, which implements the HelloOperations interface.

```
package SendMsg;
public class _tie_Hello extends SendMsg._sk_Hello {
  private SendMsg.HelloOperations _delegate;

  public _tie_Hello(SendMsg.HelloOperations delegate,
                    java.lang.String name) {
    super(name);
    this._delegate = delegate;
  }

  public _tie_Hello(SendMsg.HelloOperations delegate) {
    this._delegate = delegate;
  }

  public java.lang.String sayHello() {
    return this._delegate.sayHello();
  }
}
```

Chapter

7

The Interface Repository

This chapter describes ISB for Java's interface repository, the objects it contains, and the interfaces you can use to access it. For more information about interface repositories, see the CORBA 2.0 specification produced by the OMG at http://www.omg.org. This chapter includes the following major sections:

- Overview
- IR Structure
- Using the IR

Overview

The interface repository (IR) maintains information about ORB objects and their type. Type information is stored on objects such as modules, interfaces, operations, attributes and exceptions. An IDL interface is provided that enables client applications to query the IR to obtain language binding information or to discover newly added interfaces. The ORB accesses the IR to check the type of values in a client request or to verify an interface inheritance graph. Every object in the IR can be accessed as an `org.omg.CORBA.IRObject`, which is the most generic interface. `IRObject` provides methods for narrowing, cloning, and duplicating an `IRObject` reference. It also allows you to identify the type of an `IRObject`.

IR Structure

The IR organizes the objects it contains into a hierarchy that corresponds to the way objects are defined in an IDL specification. Some objects in the IR contain other objects, just as an IDL module definition might contain several interface definitions. Consider how the following IDL file would translate to a hierarchy of objects in the IR.

Listing 7.1 The bank/bank.idl file.

```
// Bank.idl

module Bank {
    interface Account {
        float balance();
    };
    interface AccountManager {
        Account open(in string name);
    };
};
```

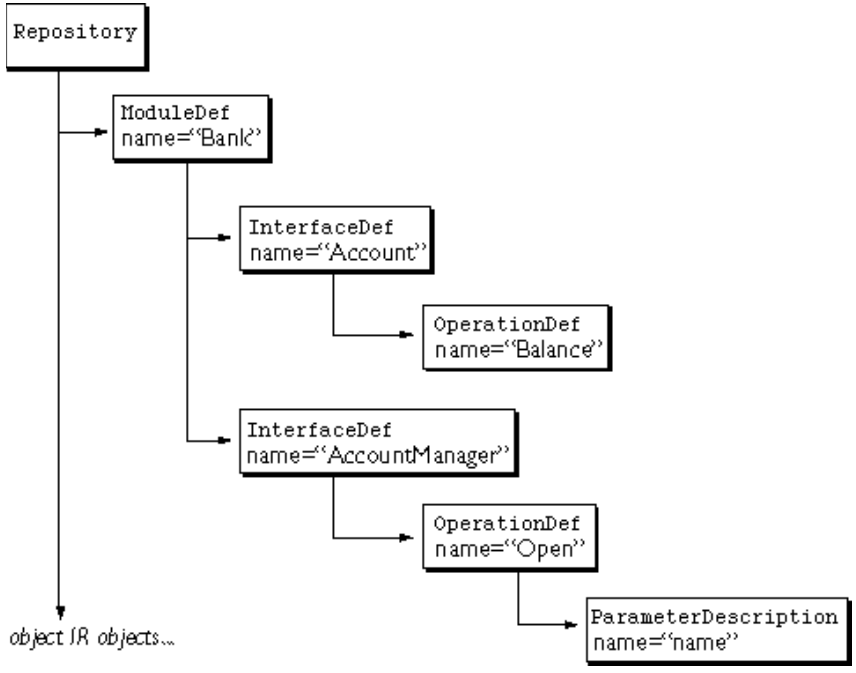

Figure 7.1 IR object hierarchy for the Bank.idl specification.

Identifying IR Objects

The following table lists objects provided to identify and classify IR objects.

Item	Description
name	A character string that corresponds to the identifier assigned in an IDL specification to a module, interface, operation, etc. An Identifier is not necessarily unique.
id	A character string that uniquely identifies an IRObject. A RepositoryID contains a sequence of identifiers, separated by "::" delimiters and always begins with a "::" delimiter.
def_kind	An enumeration that defines values which represent all the possible types of IR objects.

IRObject Types

The following table summarizes the objects that can be contained in the IR.

Object type	Description
Repository	Represents the top-level module that contains all other objects in this repository.
ModuleDef	Contains a grouping of interfaces. Can also contain constants, typedefs and even other ModuleDef objects.
InterfaceDef	Contains a list of operations, exceptions, typedefs, constants and attributes that make up an interface.
AttributeDef	Represents an attribute associated with an interface.
OperationDef	Defines an operation on an interface. It includes a list of parameters required for this operation and a list of exceptions that can be raised by this operation.
TypedefDef	A base interface for named types that are not interfaces. This includes `AliasDef`, `EnumDef`, `StructDef`, and `UnionDef`.
ConstantDef	Defines a named constant.
ExceptionDef	Defines an exception that can be raised by an operation.
IDLType	A base interface for IDL types. This includes `ArrayDef`, `PrimitiveDef`, `SequenceDef`, and `StringDef`.

Using the IR

To use the IR, do the following:

- Setting VBROKER_ADM environment variable.
- Starting the IR.
- Populating the IR.
- Accessing the IR.

Setting VBROKER_ADM

The `VBROKER_ADM` environment variable specifies the default location for the ORB log file. It enables the Interface Repository to locate the default repository files. If VBROKER_ADM is not set, the files will reside in a `\log` directory on the current drive by default.

Starting the IR

Use the `irep` command to start the IR. Specify an interface repository server and, optionally, a database containing detailed descriptions of IDL interfaces. For example:

```
prompt> irep my_ir ir_db
```

This command is described in the *Netscape Internet Service Broker for Java Reference Guide*.

Populating the IR

The `idl2ir` command can be used to populate an IR instance with objects defined in an IDL file. For example, the following command populates an IR named `my_repository` with objects defined in the file `Bank.idl`.

```
prompt> idl2ir -ir my_repository java_examples/bank/Bank.idl
```

This command is described in the *Netscape Internet Service Broker for Java Reference Guide*. You can also write your own applications that bind to an IR and add objects.

Accessing the IR

The IR offers an ORB interface that provides your client applications with a variety of methods for obtaining information about objects in the IR. Your client application can bind to the Repository and then invoke the methods shown in the Repository class (described in the *Netscape Internet Service Broker for Java Reference Guide*).

Chapter

8

Defining CORBA Interfaces With Java

ISB for Java incorporates features, collectively known as *Caffeine*, which make the product easier to work with in a Java environment. This chapter describes how to use the `java2iiop` compiler (also called the Caffeine compiler) to generate client stubs and service skeletons from interface definitions written in Java instead of IDL. This chapter also explains how to work with complex data types. In particular, it explains how to pass by value using extensible structs. This chapter includes the following major sections:

- About Caffeine
- Working with the java2iiop Compiler
- A Caffeine Example
- Mapping of Primitive Data Types
- Mapping of Complex Data Types

About Caffeine

Following are the ISB for Java features, collectively known as *Caffeine*, which ease Java development.

- The `java2iiop` compiler allows you to work in an all-Java environment. The `java2iiop` compiler takes Java interfaces and generates IIOP-compliant stubs and skeletons. One advantage of using `java2iiop` is that, through the use of extensible structs, you can pass Java serialized objects by value. This feature is unique to ISB for Java.
- The `java2idl` compiler turns Java code into IDL, allowing you to generate client stubs in the language of your choice. Also, because this compiler maps your Java interfaces to IDL, you can re-implement Java objects in another programming language that supports the same IDL. For more information about this compiler, see "Commands" in the *Netscape Internet Service Broker for Java Reference Guide*.
- The URL-based naming feature called Web Naming Service. With Web Naming, Uniform Resource Locators (URLs) can be associated with objects, allowing an object reference to be obtained by specifying a URL.

Working with the java2iiop Compiler

The `java2iiop` compiler lets you define interfaces and data types that can then be used as interfaces and data types in CORBA. The advantage is that you can define them in Java rather than IDL. The compiler reads Java bytecode: it does not read source code or java files, it reads compiled class files. The `java2iiop` compiler then generates IIOP-compliant stubs and skeletons that do the marshalling and communication required for CORBA.

When you run `java2iiop`, it generates files as if you had written the interface in IDL. Primitive data types like the numeric types (short, int, long, float, and double), String, Any, CORBA objects or interface objects, and typecodes are all understood by `java2iiop` and mapped to corresponding types in IDL.

When using `java2iiop` on Java interfaces, you define and mark them as interfaces to be used with remote calls. You mark them by having them extend the `org.omg.CORBA.Object` interface. (For developers who are familiar with RMI (Remote Method Invocations), this is analogous to a class extending the `java.rmi.Remote` interface. Caffeine provides capabilities equivalent to RMI, and allows you to create Java objects that communicate with all other objects that are CORBA-compliant, even if they are not written in Java.) The interface must also define methods that you can use in remote calls. When you run the compiler, it searches for these special CORBA interfaces. When one is found, it generates the marshalling elements (readers and writers) which enable you to use the interface for remote calls. For

classes, the compiler follows other rules and maps the classes either to IDL structs or extensible structs. For more information about complex data types, see Mapping of Complex Data Types.

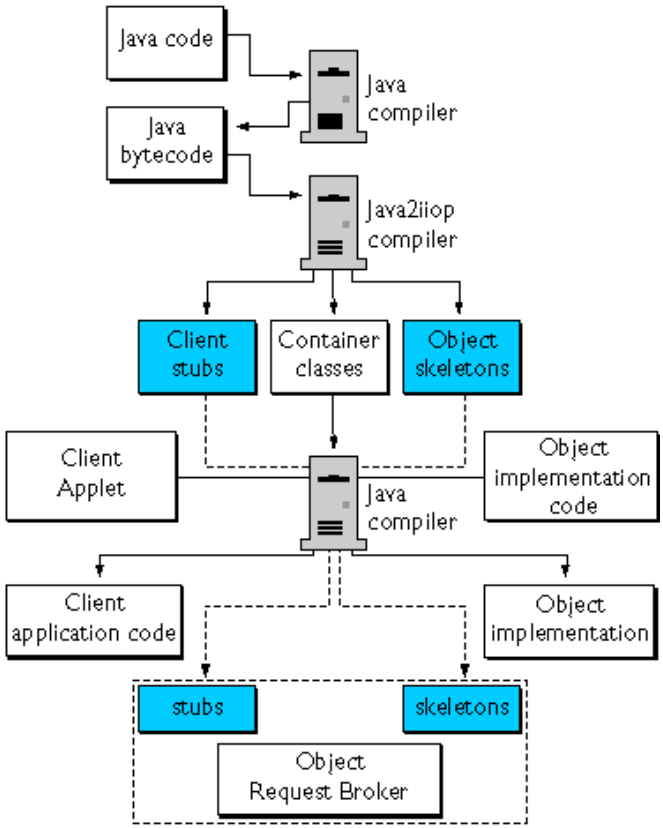

Figure 8.1 Development process when using `java2iiop`.

Running java2iiop

Before using the `java2iiop` compiler, generate Java bytecode (that is, compile a Java source file to create a class file) to input to `java2iiop`. For example, the following command compiles a Java source file named `CafMsg.java` and generates Java bytecode in a file named `CafMsg.class`.

```
prompt>javac CafMsg.java
```

After generating bytecode, you can use the `java2iiop` compiler to generate client stubs and server skeletons. The following example compiles the bytecode for `CafMsg.class`. Note that the `.class` extension is not used.

```
prompt>java2iiop CafMsg
```

The `java2iiop` compiler generates all of the usual auxiliary files such as Helper and Holder classes. For more information, see Generated Files. The files generated by the `java2iiop` compiler are the same as those generated by the `idl2java` compiler. For more information about the generated files, see "Generated Classes" in the *Netscape Internet Service Broker for Java Reference Guide*.

Completing the Development Process

After generating stubs and skeletons using the `java2iiop` compiler, create classes for the client and the service. Follow these steps:

1. Implement the service. The code is the same whether you are using Caffeine or IDL; for an example, see MsgService.java.
2. Compile the service using `javac`.
3. Write client code. The code is the same whether you are using Caffeine or IDL; for an example, see MsgClient.java.
4. Compile client code using `javac`.
5. Start the service.
6. Start the client.

A Caffeine Example

Developing with Caffeine is almost exactly the same as developing with IDL. The key difference is that you define interfaces in a Java file instead of in an IDL file. Techniques for coding implementations of interfaces, clients, and services are the same whether you are using IDL or Java.

This example begins with the Hello interface shown below. The file `Hello.java` replaces the IDL file that would define this interface if you were not using Caffeine. Mark interfaces to be used with remote calls by having them extend the `org.omg.CORBA.Object` interface.

Listing 8.1 Defining the Hello interface.

```
// Hello.java
package SendMsg;
public interface Hello extends org.omg.CORBA.Object {
    public String sayHello();
};
```

The Hello interface is implemented below in `HelloImpl.java`. The implementation is the same whether you are using Caffeine or IDL.

Listing 8.2 Hello.java implements the Hello interface.

```
// HelloImpl.java
package SendMsg;

public class HelloImpl extends SendMsg._sk_Hello {
  /** Construct a persistently named object. */
  public HelloImpl(java.lang.String name) {
    super(name);
  }

  /** Construct a transient object. */
  public HelloImpl() {
    super();
  }

  public java.lang.String sayHello() {
    // implement operation...
    System.out.println("Caf Client called sayHello.");
    return "Caf Hello!";
  }
}
```

The following code example implements the service. This code is the same whether you are using Caffeine or IDL.

```
// Using Enterprise Server's ORB
// MsgService.java
public class MsgService {
    public static void main (String[] args) {
        try {
            // Initialize the ORB.
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();
            // Initialize the BOA.
```

```
            org.omg.CORBA.BOA boa = orb.BOA_init();
            // Create the Hello object.
            SendMsg.HelloImpl hi = new SendMsg.HelloImpl("CaffeineExample");
            // Export the newly created object.
            boa.obj_is_ready(hi);
            // String host = java.net.InetAddress.getLocalHost().getHostName();
            String urlStr = "http://myHost/CaffeineExample";
            // Or, if using Communicator's ORB, the urlStr would be:
            //    "http://myHost/NameService/CaffeineExample"
            netscape.WAI.Naming.register(urlStr, hi);
            System.out.println(hi + " is ready.");
            // Wait for incoming requests
            boa.impl_is_ready();
        }
        catch(org.omg.CORBA.SystemException e) {
            System.err.println(e);
        }
    }
}
```

The following code example shows the client connecting to the service. This code is the same whether you are using Caffeine or IDL.

Listing 8.3 The client applet.

```
// Using Enterprise Server's ORB
// MsgClient.java
import netscape.WAI.Naming;

public class MsgClient extends java.applet.Applet {
    public void init() {
        try {
            // Initialize the ORB.
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();

            // Locate a message service.
            String urlStr = "http://myHost/CaffeineExample";
            // Or, if using Communicator's ORB, the urlStr would be:
            //    "http://myHost/NameService/CaffeineExample"
            org.omg.CORBA.Object obj = Naming.resolve(urlStr);
            SendMsg.Hello hi = SendMsg.HelloHelper.narrow(obj);

            // Print a message.
            System.out.println(hi.sayHello());
        }
        catch(org.omg.CORBA.SystemException e) {
            System.err.println(e);
        }
    }
```

```
    public static void main(String args[]) {
        MsgClient mc = new MsgClient();
        mc.init();
    }
}
```

To build this example, do the following:

1. `javac Hello.java`
2. `java2iiop Hello`
3. `javac HelloImpl.java`
4. `javac MsgServer.java`
5. `javac MsgClient.java`

Next, start the service in the usual way:

```
prompt> java -DDISABLE_ORB_LOCATOR MsgService
```

Finally, use one of the following techniques to start the client.

Command line	`prompt> java -DDISABLE_ORB_LOCATOR MsgClient`
AppletViewer	`prompt> appletviewer MsgClient.html`
Java-enabled browser	`http://myHost/projects/cafMsg/MsgClient.html`

Mapping of Primitive Data Types

Client stubs generated by `java2iiop` handle the marshalling of the Java primitive data types that represent an operation request so that they can be transmitted to the object server. When a Java primitive data type is marshalled, it must be converted into an IIOP-compatible format. The following table summarizes the mapping of Java primitive data types to IDL/IIOP types.

Java Type	IDL/IIOP Type
Package	Module
boolean	boolean
char	char
byte	octet

Java Type	IDL/IIOP Type
String	string
short	short
int	long
long	long long
float	float
double	double
org.omg.CORBA.Any	any
org.omg.CORBA.TypeCode	TypeCode
org.omg.CORBA.Principal	Principal
org.omg.CORBA.Object	Object

Mapping of Complex Data Types

This section discusses interfaces, arrays, Java classes, and extensible structs; it shows how the `java2iiop` compiler can be used to handle complex data types.

Interfaces

Java interfaces are represented in IDL as CORBA interfaces and they must inherit from the `org.omg.CORBA.Object` interface. When passing objects that implement these interfaces, they are passed by reference. The `java2iiop` compiler *does not* support overloaded methods on Caffeine interfaces.

Arrays

Another complex data type that can be defined in classes is an array. If you have an interface or definitions that use arrays, the arrays map to CORBA unbounded sequences.

Mapping Java Classes

In Java classes, you can define arbitrary data types. Some arbitrary data types are analogous to IDL structures (also called structs). If you define a Java class so that it conforms to certain requirements, then the `java2iiop` compiler maps it to an IDL struct. You can map Java classes to IDL structs if the class fits *all* of these requirements:

- The class is `final`.
- The class is `public`.
- The class does not use implementation inheritance.
- The data members of the class are `public`.

If a class meets all of the requirements, the compiler maps it to an IDL struct; otherwise, the compiler maps it to an extensible struct. The following example shows a simple Java class that meets all of the requirements.

Listing 8.4 An example of a Java class that would map to an IDL struct.

```
//Java
final public class Address {
   public string name;
   public string street_address;
   public short zipcode;
}
```

Extensible Structs

Any Java class that does not meet all of the requirements listed above is mapped to an extensible struct. An extensible struct is an upwardly-compatible extension of CORBA structs. When you use extensible structs, objects are passed by value.

Pass by value is the ability to pass object state to another Java program. Assuming that a class definition of the Java object is present on the server side, the Java program can invoke methods on the cloned object that has the same state as the original object.

Note The use of extensible structs is an extension to the OMG IDL: there is an additional keyword, `extensible`. If you want to stay within pure CORBA, or if you are going to port your code to other ORBs, you should use IDL structs and not extensible structs. Extensible structs allow you to use classes that can be defined in Java but cannot be defined in IDL because of CORBA limitations.

ISB uses Java serialization to pass classes in the form of extensible structs. Java serialization compresses a Java object's state into a serial stream of octets that can be passed on-the-wire as part of the request. Because of Java serialization, all the data types that are passed must be serializable; that is, they must implement `java.io.serializable`.

Extensible structs allow data that use pointer semantics to be passed successfully. For example, defining a linked list in your application code is standard practice, yet there is no way to define a linked list in standard IDL. The solution to this problem is that you can define it by using extensible structs. When you pass extensible structs across address spaces, pointers are maintained.

An Extensible Struct Example

This section providing several code samples showing how to use extensible structs. The extensible struct code samples show the ability to pass these data types (extensible structs) and that, when passed, they retain their values. They behave like any other CORBA data type and they have an additional advantage: with extensible structs you can go beyond what you can do with IDL and pass arbitrary Java serializable objects.

The code samples provided here are simple ones; you could write a much more complicated data structure that has a tree, or a complicated linked list, or a skip list. Using extensible structs, you could pass any of these complicated data structures by value. The code sample below shows a fairly simple class that cannot be defined in IDL. The class has a constructor to help construct linked lists and a `toString` method which prints out the values of the list. With the mapping rules in mind, there are a few things which point out that this class will map as an extensible struct:

- The class is not `final`.
- As all extensible structs must do, it implements `java.io.Serializable`.
- It has a self-referential data member: a class called List and a data member called List. This is not possible in IDL.
- It has private data members.

Listing 8.5 An example of a Java class that would be mapped as an extensible struct.

```
// List.java
public class List implements java.io.Serializable {
  private String _data;
  private List _next;
```

```
  public List(String data, List next) {
    _data = data;
    _next = next;
  }

  public String data() {
    return _data;
  }

  public List next() {
    return _next;
  }

  public void next(List next) {
    _next = next;
  }

  public String toString() {
    StringBuffer result = new StringBuffer("{ ");
    for(List list = this; list != null; list = list.next()) {
      result.append(list.data()).append(" ");
    }
    return result.append("}").toString();
  }
}
```

Next is the ListUtility interface, shown below. The interface has all the attributes of a Caffeine-enabled interface: It is a public interface that extends `org.omg.CORBA.Object`. The ListUtility interface defines these methods:

- `length` - This method computes the length of the list.
- `reverse` - This method returns a list that is in reverse order of items that were used as input.
- `sort` - This method sorts the list.

Listing 8.6 A Caffeine-enabled interface.

```
// ListUtility.java
public interface ListUtility extends org.omg.CORBA.Object {
  public int length(List list);
  public List reverse(List list);
  public List sort(List list);
}
```

The ListServer class, shown below, is a standard server class. It extends the skeleton, has a constructor, and has implementations of all of the methods (`length`, `reverse`, and `sort`). However, the List implementations in these code samples do not support circular lists.

The `length` and `reverse` methods are straightforward. The `sort` method is implemented by doing a merge sort. The `sort` method has a base case which calculates that if the length of the list is one or less, the list is already sorted. Otherwise, it splits the list in half, sorts each half of the list, and merges the two halves into a sorted list. To do the merge portion of the merge sort, the `merge` method takes two sorted lists and merges them into a single sorted list.

The `main` method in ListServer is standard: it initializes the ORB and BOA, instantiates ListServer, gets the object ready, and prints out that the object is ready.

Listing 8.7 Implementing the List interface.

```
// ListServer.java
import org.omg.CORBA.ORB;
import org.omg.CORBA.BOA;

public class ListServer extends _sk_ListUtility {
  ListServer(String name) {
    super(name);
  }

  public int length(List list) {
    int result = 0;
    for( ; list != null; list = list.next()) {
      result++;
    }
    return result;
  }

  public List reverse(List list) {
    List result = null;
    for( ; list != null; list = list.next()) {
      result = new List(list.data(), result);
    }
    return result;
  }

  public List sort(List list) {
    int length = length(list);
    if(length <= 1) {
      return list;
    }
```

```
    // split the list in half
    List lhs = list;

    for(int i = 0; i < length / 2 - 1; i++) {
      list = list.next();
    }

    List rhs = list.next();
    // this actually splits the lists
    list.next(null);
    // sort and merge the two halves
    return merge(sort(lhs), sort(rhs));
  }

  public List merge(List lhs, List rhs) {
    if(lhs == null) {
      return rhs;
    }

    if(rhs == null) {
      return lhs;
    }

    // figure out which side is next
    List head;
    if(lhs.data().compareTo(rhs.data()) < 0) {
      head = lhs;
      lhs = lhs.next();
    }
    else {
      head = rhs;
      rhs = rhs.next();
    }

    head.next(merge(lhs, rhs));
    return head;
  }

  public static void main(String[] args) {
    ORB orb = ORB.init();
    BOA boa = orb.BOA_init();
    ListUtility impl = new ListServer("demo");
    boa.obj_is_ready(impl);
    System.out.println(impl + " is ready.");
    boa.impl_is_ready();
  }
}
```

The ListClient is also standard: it binds to the ListServer, in three separate short sections it creates lists, and, finally it calls the operations on each list. The first section prints out a list called "Hello World". It prints out the length, reverse order, and the sorted version. To view an example of the output, see below. The next section is a slightly more complicated list--"This is a test"--and it does the same thing as the first list. The third section in the code reads in the file for `ListUtility.java`. It reads it into a string and then breaks it into tokens. It breaks on punctuation and white space. It, too, prints the length, reverse order, and sorted version.

Listing 8.8 The Client Application for the list.

```
// ListClient.java
import org.omg.CORBA.ORB;
import java.io.*;
import java.util.*;

public class ListClient {
  public static void main(String[] args) throws Exception {
    ORB orb = ORB.init();
    String host = args[0];
    ListUtility lu =
        ListUtilityHelper.narrow
            (netscape.WAI.Naming.resolve("http://" + host + "/listutil"));
    {
      List list = new List("Hello", new List("World", null));
      System.out.println("list:    " + list);
      System.out.println("length:  " + lu.length(list));
      System.out.println("reverse: " + lu.reverse(list));
      System.out.println("sort:    " + lu.sort(list));
    }

    {
      List list = new List("this",
                  new List("is",
                  new List("a",
                  new List("test", null))));
      System.out.println("list:    " + list);
      System.out.println("length:  " + lu.length(list));
      System.out.println("reverse: " + lu.reverse(list));
      System.out.println("sort:    " + lu.sort(list));
    }

    {
      // read in the contents of a file
      InputStream input = new FileInputStream("ListUtility.java");
      byte[] bytes = new byte[input.available()];
      input.read(bytes);
      String text = new String(bytes);
      List list = null;
```

```
      // break the input into tokens (ignoring white space and punctuation)
      StringTokenizer tokenizer = new StringTokenizer(text, " \n.;,/{}()");
      while(tokenizer.hasMoreTokens()) {
        list = new List(tokenizer.nextToken(), list);
      }

      System.out.println("list:    " + list);
      System.out.println("length:  " + lu.length(list));
      System.out.println("reverse: " + lu.reverse(list));
      System.out.println("sort:    " + lu.sort(list));
    }
  }
}
```

To build this example, do the following:

1. `javac ListUtility.java`
2. `java2iiop ListUtility`
3. `javac ListClient.java`
4. `javac ListServer.java`

Here is the output.

Listing 8.9 The List output.

```
list:    { Hello World }
length:  2
reverse: { World Hello }
sort:    { Hello World }
list:    { this is a test }
length:  4
reverse: { test a is this }
sort:    { a is test this }
list:    { list List sort List public list List reverse List public list List
length int public Object CORBA omg org extends ListUtility interface public
java ListUtility }
length:  25
reverse: { ListUtility java public interface ListUtility extends org omg CORBA
Object public int length List list public List reverse List list public List
sort List list }
sort:    { CORBA List List List List List ListUtility ListUtility Object
extends int interface java length list list list omg org public public public
public reverse sort }
```

Extensible Structs and GIOP Messages

The following section is a discussion of an advanced topic and is provided for developers writing code for non-ISB ORBs who want their code to understand and interpret extensible structs as implemented in ISB. Please be advised that if GIOP formats change, the way ISB works with GIOP messages will also change and your applications may need to be modified.

When messages are sent with an interface that extends `org.omg.CORBA.Object`, a GIOP (General Inter-ORB protocol) message is created. The standard header is included in the message, as well as the arguments. Each argument is put into an octet sequence containing the Java serialization. An octet sequence is a standard CORBA type used in GIOP messages to pass complex data types. If you are working with communication protocols and understand Java serialization, you can take this grouping of bytes in the GIOP message and extract from it the arguments being passed.

Since an extensible struct is just a stream of octets (that is, an array of bytes), a developer working with a non-Netscape ORB can write code that does the following if an extensible struct is passed to that non-Netscape ORB:

- Interprets the contents of an extensible struct.
- Stores the stream of octets and later passes it on to another ORB.

Chapter

9

Managing Threads and Connections

This chapter discusses the use of threads and thread policy in client applications and object implementations. Java applications, by their very nature, support multiple threads and the ISB for Java core automatically uses threads for its internal processing, resulting in more efficient request management. All code within a server that implements an ORB object must be thread-safe. You must take special care when accessing a system-wide resource within an object implementation. For example, many database access methods are not thread-safe. Before your object implementation attempts to access such a resource, it must first lock access to the resource using a synchronized block.

Server Thread-per-session Policy

ISB for Java supports a *thread-per-session* policy for managing threads. With the thread-per-session policy, threading is driven by connections between the client and server processes. A new worker thread is allocated each time a client connects to a server. A worker thread is assigned to handle all the requests received from a particular client. When the client disconnects from the server, the worker thread is destroyed.

Figure 9.1 shows the use of the thread-per-session policy. The Client Application #1 establishes a connection with the object implementation. A separate connection exists between Client Application #2 and the object implementation. When a request comes in to the object implementation from Client Application #1, a worker thread handles the request. When a request from Client Application #2 comes in, a different worker thread is assigned to handle this request.

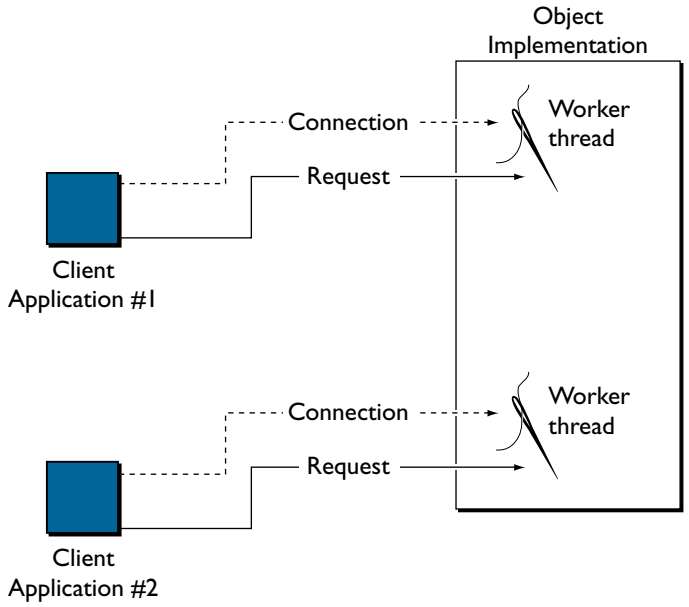

Figure 9.1 An object implementation using the thread-per-session policy

In Figure 9.2, a second request has come in to the object implementation from Client Application #1. The same worker thread that handles request 1 will handle request 2. When the worker thread completes request 1, then it can handle request 2 from Client Application #1--and any other future requests that come in after it finishes. Multiple requests can come in from Client Application #1--they are handled in the order that they come in; no additional worker threads are assigned to Client Application #1.

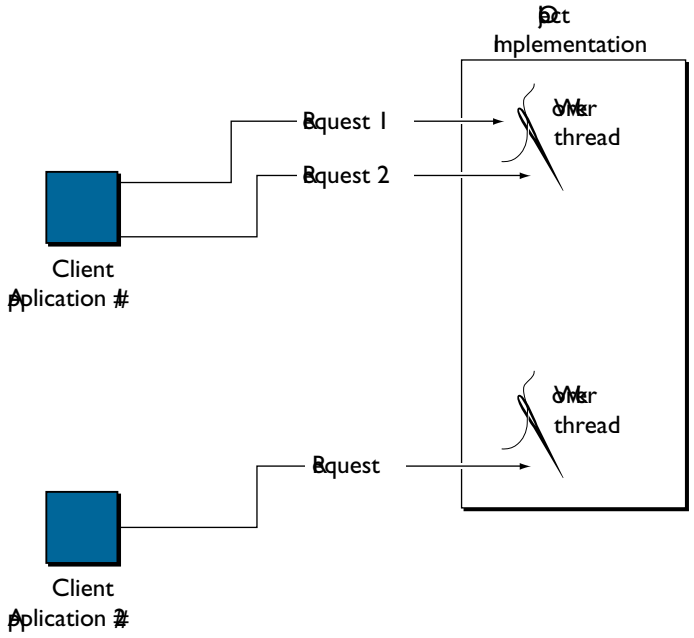

Figure 9.2 Thread-per-session: a second request comes in from the same client

If you want the worker thread to handle more than one request from the same Client Application simultaneously, you must call the `Object._clone` method. This creates a second connection from the Client Application to the Object Implementation and enables the Object Implementation to handle another request simultaneously. If the client uses the `_clone` operation, another connection is established. Because connections are a limited operating resource, it is important to limit the use of `_clone` operations.

Chapter

10

The Dynamic Skeleton Interface

This chapter describes how object servers can dynamically create object implementations at run time to service client requests. The Dynamic Skeleton Interface (DSI) allows an object to register itself with the ORB, receive operation requests from a client, process the requests, and return the results to the client without inheriting from a skeleton class. This chapter includes the following major sections:

- Overview
- Using DSI
- The DynamicImplementation Class
- The ServerRequest Class
- Implementing the Account Object
- Implementing the AccountManager Object
- The Server Implementation

Overview

The *Dynamic Skeleton Interface* (DSI) provides a mechanism for creating an object implementation that does not inherit from a generated skeleton interface. Normally, an object implementation is derived from a skeleton class generated by the `idl2java` compiler. DSI allows an object to register itself with the ORB, receive operation requests from a client, process the requests, and return the results to the client without inheriting from a skeleton class. From the perspective of a client application, an object implemented with DSI behaves just like any other ORB object. Clients do not need to provide any special handling for object implementations that use DSI.

The ORB presents a client operation request to a DSI object implementation by using the object's invoke method and passing a ServerRequest object. The object implementation is responsible for determining the operation being requested, interpreting the arguments associated with the request, invoking the appropriate internal method or methods to fulfill the request, and returning the appropriate values.

Implementing objects with DSI requires more manual programming activity than using the normal language mapping provided by object skeletons. Nevertheless, an object implemented with DSI can be very useful in providing inter-protocol bridging or in allowing one object to offer multiple interfaces.

Example Program

This example is used to illustrate DSI concepts in this chapter. The example uses the Bank.idl file shown below.

Listing 10.1 The Bank.idl file used in the DSI example.

```
// Bank.idl
module Bank {
    interface Account {
        float balance();
    };
    interface AccountManager {
        Account open(in string name);
    };
};
```

Using DSI

To use DSI for your object implementation, follow these steps:

1. Instead of extending a skeleton object, design your object implementation so that it extends the `org.omg.CORBA.DynamicImplementation` abstract class.
2. Declare and implement the `invoke` method, which the ORB will use to dispatch client requests to your object.
3. Register your object implementation with the BOA in the normal way, using the `boa.obj_is_ready` method.

The DynamicImplementation Class

Any object implementations that you wish to use DSI should be derived from the DynamicImplementation base class. This class offers several constructors and the invoke method, which you, the programmer, must implement. For complete details on this class, see the *Netscape Internet Service Broker for Java Reference Guide*.

Listing 10.2 The DynamicImplementation base class.

```
abstract public class DynamicImplementation extends
org.omg.CORBA.portable.Skeleton {

    abstract void invoke(ServerRequest request);

    protected DynamicImplementation(String object_name,
                                    String repository_id) {
        ...
    }
    protected DynamicImplementation(String object_name,
                                    String[] repository_ids) {
        ...
    }

    ...
}
```

Thefollowingcodeexampleshowsthedeclarationof theAccountandAccountManager objects that are to be implemented with DSI. Both are derived from the DynamicImplementation class, which adds the invoke method. The ORB uses the invoke method to pass client operation requests to the object in the form of ServerRequest objects.

Listing 10.3 A portion of Server.java from the DSI example.

```
class Account extends org.omg.CORBA.DynamicImplementation {
    Account(String name) {
        super(name, "IDL:Bank/Account:1.0");
    }
    void invoke(org.omg.CORBA.ServerRequest request) {
    }
    ...
}

class AccountManager extends org.omg.CORBA.DynamicImplementation {
    Account(String name) {
        super(name, "IDL:Bank/AccountManager:1.0");
    }
    void invoke(org.omg.CORBA.ServerRequest request) {
    }
    ...
}
```

Specifying Repository Ids

The first form of the DynamicImplementation class' constructor allows you to create a server object, specifying a string that represents the repository identifier of the object that is being implemented. Note that both the Account and AccountManager classes declare a constructor that specifies the appropriate repository identifier. To determine the correct repository identifier to specify, start with the IDL interface name of an object and use the following steps:

1. Replace all non-leading instances of the delimiter "::" with "/".
2. Add "IDL:" to the beginning of the string.
3. Add ":1.0" to the end of the string.

For example, given an IDL interface name of Bank::AccountManager, the resulting repository identifier string is `IDL:Bank/AccountManager:1.0`.

The ServerRequest Class

A ServerRequest object is passed as a parameter to an object implementation's `invoke` method. The ServerRequest object represents the operation request and provides methods for obtaining the name of the requested operation, the parameter list, and the context. It also provides methods for setting the result to be returned to the caller and for reflecting exceptions. This class is described in the *Netscape Internet Service Broker for Java Reference Guide*.

Listing 10.4 The ServerRequest class.

```
abstract public class ServerRequest {
    abstract public java.lang.String op_name();
    abstract public org.omg.CORBA.Context ctx();
    abstract public void params(
        org.omg.CORBA.NVList params
    );
    abstract public void result(
        org.omg.CORBA.Any result
    );
    abstract public void except(
        org.omg.CORBA.Any except
    );
}
```

Implementing the Account Object

The Account object in this example offers only one method, so the processing done by its `invoke` method is fairly straightforward.

The `invoke` method first checks to see if the requested operation has the name "balance." If the name does not match, a BAD_OPERATION exception is raised. If the Account object were to offer more than one method, the `invoke` method would need to check for all possible operation names and use the appropriate internal methods to process the operation request.

Since the `balance` method does not accept any parameters, there is no parameter list associated with its operation request. The balance method is simply invoked and the result is packaged in an Any object that is returned to the caller, using the `ServerRequest.result` method.

Listing 10.5 The Account.invoke method.

```
class Account extends org.omg.CORBA.DynamicImplementation {
    Account(float balance) {
        super(null, "IDL:Bank/Account:1.0");
        _balance = balance;
    }
    public void invoke(org.omg.CORBA.ServerRequest request) {
        // make sure the operation name is correct
        if(!request.op_name().equals("balance")) {
            throw new org.omg.CORBA.BAD_OPERATION();
        }
        // no parameters, so simply invoke the balance operation
        float balance = this.balance();
        // create an Any for the result
        org.omg.CORBA.Any balanceAny = _orb().create_any();
        // put in the balance,
        balanceAny.insert_float(balance);
        // set the request's result
        request.result(balanceAny);
    }
    float balance() {
        return _balance;
    }
    private float _balance;
}
```

Implementing the AccountManager Object

Like the Account object, the AccountManager offers only one method. However, the `AccountManager.open` method does accept an account name parameter. This makes the processing done by the `invoke` method a little more complicated. The following code example shows the implementation of the `AccountManager.invoke` method.

The method first checks to see that the requested operation has the name "open." If the name does not match, a BAD_OPERATION exception is raised. If the AccountManager object were to offer more than one method, the invoke method would need to check for all possible operation names and use the appropriate internal methods to process the operation request.

Listing 10.6 The AccountManager.invoke method.

```
class AccountManager extends org.omg.CORBA.DynamicImplementation {
    AccountManager(String name) {
        super(name, "IDL:Bank/AccountManager:1.0");
    }
    public void invoke(org.omg.CORBA.ServerRequest request) {
```

```
        // make sure the operation name is correct
        if(!request.op_name().equals("open")) {
            throw new org.omg.CORBA.BAD_OPERATION();
        }
        // create an empty parameter list
        org.omg.CORBA.NVList params = _orb().create_list(0);
        // create an Any for the account name parameter
        org.omg.CORBA.Any nameAny = _orb().create_any();
        // set the Any's type to be a string
        nameAny.type(_orb().get_primitive_tc
                            (org.omg.CORBA.TCKind.tk_string));
        // add "in" the parameter to the parameter list
        params.add_value("name", nameAny, org.omg.CORBA.ARG_IN.value);
        // obtain the parameter values from the request
        request.params(params);
        // get the name parameter from the Any
        String name = nameAny.extract_string();
        // invoke the open operation
        Account account = this.open(name);
        // create an Any for the result
        org.omg.CORBA.Any accountAny = _orb().create_any();
        // put the new Account object into an Any
        accountAny.insert_Object(account);
        // set the request's result
        request.result(accountAny);
    }
    public Account open(String name) {
        ...
    }
}
```

Processing Input Parameters

Here are the steps that the `AccountManager.invoke` method uses to process the operation request's input parameters.

1. Create an NVList to hold the parameter list for the operation.
2. Create Any objects for each expected parameter and add them to the NVList, setting their TypeCode and parameter type (ARG_IN or ARG_INOUT).
3. Invoke the `ServerRequest.param` method, passing the NVList, to update the values for all the parameters in the list.

Since the `open` method expects an account name parameter, an NVList object is created to hold the parameters contained in the ServerRequest. The NVList class implements a parameter list containing one or more NamedValue object. The NVList and NamedValue classes are described in The Dynamic Invocation Interface and in the *Netscape Internet Service Broker for Java Reference Guide*.

An Any object is created to hold the account name. This Any is then added to NVList with the argument's name set to "name" and the parameter type set to ARG_IN.

Once the NVList has been initialized, the ServerRequest.params method is invoked to obtain the values of all of the parameters in the list.

After invoking the params method, the NVList will be owned by the ORB. This means that if an object implementation modifies an ARG_INOUT parameter in the NVList, the change will automatically be apparent to the ORB.

An alternative to constructing the NVList for the input arguments is to use the `org.omg.CORBA.ORB.create_operation_list` method. This method accepts an OperationDef and returns an NVList object, completely initialized with all the necessary Any objects. The appropriate OperationDef object can be obtained from the Interface Repository, described in Chapter 7.

Setting the Return Value

After invoking the `ServerRequest.params` method, the account name value can be extracted and used to create a new Account object. An Any object is created to hold the newly created Account object, which is returned to the caller by invoking the `ServerRequest.result` method.

The Server Implementation

The implementation of the Server class, shown below, is the same whether or not you are using DSI.

Listing 10.7 The Server class implementation.

```
// Using Enterprise Server's ORB
public class Server {
    public static void main(String[] args) {
        try {
            // Initialize the ORB.
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();
```

```
            // Initialize the BOA.
            org.omg.CORBA.BOA boa = orb.BOA_init();
            // Create the account manager object.
            AccountManager manager = new AccountManager("ISB_Bank");
            // Export the newly create object.
            boa.obj_is_ready(manager);
            String host =
                java.netInetAddress.getLocalHost().getHostName();
            netscape.WAI.Naming.register
                ("http://" + host + "/ISB_Bank", manager);

            // Or, if using Communicator's ORB, the call would be:
            // netscape.WAI.Naming.register
            //   ("http://" + host + "/NameService/ISB_Bank", manager);

            System.out.println(manager + " is ready.");
            // Wait for incoming requests
            boa.impl_is_ready();
        }
        catch(org.omg.CORBA.SystemException e) {
            System.err.println(e);
        }
        catch(java.net.UnknownHostException u) {
            System.err.println(u);
        }
    }
}
```

Appendix

A

Troubleshooting

This appendix explains how to troubleshoot common build and compilation errors by checking the settings of system properties and environment variables and debugging the runtime. It includes these major sections:

- Language Mapping and Name Changes
- Build Errors
- Compilation Errors
- Debugging the Runtime
- System Properties
- Environment Variables

Language Mapping and Name Changes

The OMG IDL to Java language mapping specification has changed, causing IDL to Java language mapping changes. ISB for Java conforms with the *OMG IDL/Java Language Mapping Specification*. A brief list of differences between the new language mapping and the old language mapping follows:

- The CORBA package has been renamed to `org.omg.CORBA`.

- IDL `enum` types map to instances of a Java classes with the same names as the IDL type. This allows enumerated types to be used in a type-safe manner.
- The exception model has changed. CORBA system exceptions have been made a subclass of Java runtime exceptions. This means that CORBA system exceptions do *not* need to be declared in all method signatures which might raise such exceptions.
- The *type*_var class has been broken into two classes, *type*Holder and *type*Helper. The Holder class is used to pass `inout` and `out` parameters. The Helper class is used to hold all the auxiliary methods relating to the type (for example, the static method to insert the type into an Any).
- The `org.omg.CORBA.Any` class has been significantly modified. In particular, the class is now abstract. Instances of the class are obtained via the `org.omg.CORBA.ORB.create_any` method. All of the methods to insert and extract data from the Any have been changed (for example, `to_string` and `from_string` are now `insert_string` and `extract_string`).

Because of these language mapping changes, the following conversions may be necessary in your code:

- Break up the *type*_var class into two classes, *type*Holder and *type*Helper.
- Change `CORBA` to `org.omg.CORBA`.
- Because CORBA system exceptions do not need to be declared in all method signatures, remove "throws SystemException".
- Change booleanHolder to BooleanHolder
- Change byteHolder to ByteHolder
- Change charHolder to CharHolder
- Change doubleHolder to DoubleHolder
- Change floatHolder to FloatHolder
- Change intHolder to IntHolder
- Change longHolder to LongHolder
- Change shortHolder to ShortHolder

- Convert all references from `CORBA.Object` to `org.omg.CORBA.Object`, and remove all references to the class `CORBA.Exception`.
- The static `bind` operations on the Helper class now require that an instance of `org.omg.CORBA.ORB` be supplied.

Name Changes

If you are upgrading from a previous version of Netscape ISB for Java or Visigenic VisiBroker for Java, you may encounter the name changes listed in the following table.

Type	Old Name	New Name
Environment Variable	ORBELINE	VBROKER_ADM
	ORBELINE_IMPL_PATH	VBROKER_IMPL_PATH
Directory	orb2.0 or /Orbeline2.0	vbroker or /vbroker
Commands	bw	java
Classes	pomoco	com.visigenic.vbroker

Build Errors

This section lists errors that may be encountered while building the example applications and the solutions to those problems.

The executable idl2java is not in your path.

Typical output:

```
prompt> make idl
idl2java Bank.idl
sh: idl2java: not found
*** Error code 1
make: Fatal error: Command failed for target 'idl'
```

Solutions:

- Add the `[server]/wai/bin` directory to your path, where `[server]` represents the Enterprise Server root.
- Modify the Makefile to use the full path of the `idl2java` executable.

Need to run a preprocessor on UNIX input files.

Before using `idl2java` to compile UNIX format files you must invoke the C preprocessor manually. Then you can run `idl2java` on the preprocessed file. As an alternative, you can convert UNIX format files to DOS format, then run `idl2java`.

The java compiler (javac) or the java interpreter (java) are not in your path

Typical output:

```
prompt> make javac Client.java
sh: javac: not found
*** Error code 1
make: Fatal error: Command failed for target `Client.class'
```

Solutions:

- Add the Java `bin` directory to your path and rehash.
- Modify the Makefile to use the full path of the `javac` binary.

U Substitute<installation_location>withthelocationwhereyouinstalledISBforJava and, assuming you are using C shell, update your path like the following:

```
setenv PATH /<installation_location>/java/bin:$PATH
```

Compilation Errors

Therearenumerouscompilationerrorsthatusershaveencountered.Pleaselookthrough the outputs and see if anything resembles what you are encountering.

Typical output (truncated):

```
prompt> make
```

```
javac Client.java
Client.java:7: Undefined variable: org
org.omg.CORBA.ORB.init();
^
./Bank/AccountManager.java:2: Interface org.omg.CORBA.Object of
interface Bank.AccountManager not found.
public interface AccountManager extends org.omg.CORBA.Object
^
```

Solutions:

- Set the Java environment variable CLASSPATH. Typically, this variable contains:
  - The current working directory
  - The classes directory of the ISB for Java distribution
  - The classes directory of the Java distribution

U Substitute <installation_location> with the location where you installed both Java and ISB for Java and assume you are using C shell. You would enter the following:

```
setenv CLASSPATH .:/<installation_location>/java/classes:/<installation_location>/isb/
classes
```

Recompiling IDL Files

The runtime and code generators have been upgraded to be compatible with JDK 1.1.

To be compatible with the new runtime, you must use the `idl2java` compiler and recompile all IDL files. This will create new stub, skeleton, and support files which are compatible with the *OMG IDL /Java Language Mapping Specification*.

Make sure you make backups of all the IDL files before you use `idl2java`.

Debugging the Runtime

The ISB for Java runtime can be configured to provide detailed diagnostics of its functionality. The debugging diagnostics can be controlled:

- From the command line

- From an HTML file specifying the Applet

The runtime diagnostics can be turned on from the command line by specifying the CORBA_DEBUG property as follows:

```
prompt> java -DCORBA_DEBUG Client
...
```

The runtime diagnostics can be turned on via HTML by specifying the CORBA_DEBUG parameter as follows:

```
<applet code=ClientApplet.class width=200 height=80>
<param name=CORBA_DEBUG value=1>
</applet>
```

System Properties

Various problems can be solved by checking the setting of System Properties. Use this section as a guide to setting System Properties appropriately for your needs and environment.

To set system properties, use the java command. Always add a -D before the system property, and the system property must appear before the Java class name.

CORBA_DEBUG

The CORBA runtime diagnostics can be turned on by specifying the CORBA_DEBUG property as follows:

```
prompt> java -DCORBA_DEBUG Client
...
```

OAport

Specifies the port number to be used by a server. If a port number is not specified, an unused port will be selected.

Environment Variables

Various problems can be solved by checking the setting of environment variables. Use this section as a guide to setting environment variables appropriately for your needs and environment.

CLASSPATH

Set the CLASSPATH environment variable to tell the Java interpreter where to look for your classes. Since you are using ISB for Java classes, the CLASSPATH must include the path to these classes. By default, classes reside in the following directory:

```
c:\netscape\suitespot\wai\java\nisb.zip
```

PATH

Set your PATH environment variable to include the bin directory of the ISB for Java distribution and to the directory where the Java binaries have been installed.

VBROKER_ADM

For Windows users, the default location for ORB log files will be the directory pointed to by the VBROKER_ADM environment variable. If this variable is not set, they will be found in the `log` directory on the current drive.

VBROKER_IMPL_NAME

Set this to change the directory name where the implementation repository files reside. Setting this environment variable overrides IMPL_DIR.

VBROKER_IMPL_PATH

Defines the path to the directory where implementation repository files reside. Setting this environment variable overrides VBROKER_ADM.