orb2 for Java
User’s Guide
Release 6.0
Subject

Instructions for developing applications with orb2 for Java

Software Supported

orb2 for Java 6

Revision History

<table>
<thead>
<tr>
<th>Release</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release 2.1 Product Name Change</td>
<td>April 2000</td>
</tr>
<tr>
<td>Content updates and additions</td>
<td>November 2000</td>
</tr>
<tr>
<td>Rebranded</td>
<td>September 2001</td>
</tr>
<tr>
<td>Release 2.3</td>
<td>September 2002</td>
</tr>
<tr>
<td>Release 4.0</td>
<td>April 2003</td>
</tr>
<tr>
<td>Patch Release 4.0.4</td>
<td>August 2003</td>
</tr>
<tr>
<td>Release 4.1</td>
<td>September 2004</td>
</tr>
<tr>
<td>Release 4.2</td>
<td>May 2005</td>
</tr>
<tr>
<td>Release 5.0</td>
<td>November 2007</td>
</tr>
<tr>
<td>Release 5.1</td>
<td>December 2008</td>
</tr>
<tr>
<td>Release 6.0</td>
<td>July 2013</td>
</tr>
</tbody>
</table>
2AB, Inc. disclaims the implied warranties of
merchantability and fitness for a particular purpose and
makes no express warranties except as may be stated in its
written agreement with and for its customer. In no event
is 2AB, Inc. liable to anyone for any indirect, special or
consequential damages.

The information and specifications in this document are
subject to change without notice. Consult your 2AB, Inc.
marketing representative for product or service
availability.

U.S. Government Restricted Rights. The Software
Program(s) and Documentation furnished under this
Agreement were developed at private expense and are
provided with Restricted Rights. Any use, duplication, or
disclosure by and for any agency of the U.S. Government
shall be subject to the Restricted Rights applicable to
commercial computer software under FAR Clause
52.227-19 or DFAR Clause 252.277-7013 or any
successor thereof.

Copyright © 1999-2005 by 2AB, Inc.
All Rights Reserved.

Trademarks

The 2AB logo is a registered trademark of 2AB, Inc.
iLock, iLock Security Services, jLock, orb2, orbLock,
and webLock are trademarks of 2AB, Inc

Microsoft, Windows(all versions), including but not
limited to Windows XP, VISTA, Window 7, Windows 8,
and Windows Server 200x are registered trademarks of
the Microsoft Corporation.

OMG, Object Management Group, OMG Interface
Definition Language (IDL), Unified Modeling Language
and UML are trademarks of the Object Management
Group. CORBA and IIOP are registered trademarks of the
Object Management Group.

UNIX is a registered trademark in the United States and
other countries, licensed exclusively through X/Open
Company Limited.

HP-UX is a registered trademarks of the Hewlett-Packard
Company.

Apache and Tomcat are trademarks of The Apache
Software Foundation.

AIX and Domino are trademarks of the International
Business Machines Corporation in the United States or
other countries or both.

iPlanet, J2EE, J2SE, Java, JavaScript, JavaServer Pages,
JDK, Solaris, Sun and Sun Microsystems are trademarks
or registered trademarks of Oracle Inc. in the United
States and other countries.

All other brand or product names are trademarks or
registered trademarks of their respective companies or
organizations.
## Contents

### Chapter 1  Getting Started
1.1 CORBA Interoperability .................................................. 1-1  
1.2 Development Environment ............................................ 1-1  
1.3 Runtime Environment .................................................. 1-1  
1.4 Licensing ................................................................. 1-1  
1.5 Environmental Variables .............................................. 1-1  
1.6 Classpath and System Properties ................................... 1-2  
1.7 Locating Objects ....................................................... 1-2  
1.7.1 Stringified Object References ................................... 1-2  
1.7.2 Naming Service ...................................................... 1-3  
1.7.3 Trader Service ....................................................... 1-3  
1.8 Demonstration Programs .............................................. 1-3  

### Chapter 2  Developing a Distributed orb2 Application
2.1 Interfaces ............................................................... 2-1  
2.2 Compiling IDL .......................................................... 2-2  
2.3 Developing Server Programs ...................................... 2-5  
2.3.1 Developing the Calculator Server Program ..................... 2-6  
2.4 Developing Client Programs ....................................... 2-9  
2.4.1 Developing the Calculator Client Program ..................... 2-10  

### Chapter 3  Compiling IDL
3.1 Introduction ............................................................ 3-1  
3.2 The idlc IDL Compiler ................................................ 3-1  
3.2.1 Running the idlc Compiler ........................................ 3-1  
3.2.1.1 General Options .................................................. 3-1  
3.2.1.2 Package Name Options ......................................... 3-2  
3.2.1.3 Stub/Skeleton Generation Options ............................ 3-4  
3.2.2 Running Without the idlc Script ................................ 3-4  
3.2.3 OpenOrb Copyright ................................................ 3-4  
3.3 The idl2wsdl IDL Compiler ......................................... 3-5  
3.3.1 Running the idl2wsdl Compiler ................................... 3-5  
3.3.1.1 General Options .................................................. 3-5  
3.3.1.2 Package Name Options ......................................... 3-6  
3.3.1.3 WSDL options ..................................................... 3-6  
3.3.2 Running Without the idl2wsdl Script ........................... 3-6  

### Chapter 4  Initializing the ORB
4.1 Writing the Initialization Software ................................ 4-1
## Contents

4.2 ORB Property Files ......................................................... 4-1  
4.3 Precedence of Properties ................................................. 4-2  
4.4 Command-Line Arguments. ................................................. 4-2  
  4.4.1 General Purpose .................................................... 4-2  
  4.4.2 Security ............................................................... 4-4  
  4.4.3 Trader Service ....................................................... 4-4  
4.5 ORB Properties ............................................................ 4-5  
  4.5.1 General Purpose Properties .......................................... 4-5  
  4.5.2 Security ............................................................... 4-7  
  4.5.3 Trader Service ....................................................... 4-11  

Chapter 5 orb2 Security Features 
5.1 Common Secure Interoperability Version2 (CSIv2) ...................... 5-1  
5.2 CORBA Security Services (CSS) ......................................... 5-1  
5.3 Resource Access Decision Facility ....................................... 5-1  
5.4 Auditing ........................................................................ 5-2  
5.5 Programming API's ......................................................... 5-2  
5.6 The Security Center ......................................................... 5-2  
5.7 Identity and Policy Administration Tools .................................. 5-2  
5.8 orb2 Security Manuals ...................................................... 5-2  
  5.8.1 orb2 Security Services User Guide .................................. 5-2  
  5.8.2 Resource Access Decision Facility User Guide .................... 5-2  

Chapter 6 Portable Object Adapter 
6.1 Creating a POA ............................................................... 6-2  
  6.1.1 Using the Root POA ..................................................... 6-3  
  6.1.2 Using a Custom POA ................................................... 6-4  
6.2 Servant Objects .............................................................. 6-6  
  6.2.1 Servant Managers ....................................................... 6-8  
    6.2.1.1 ServantActivator Interface .................................... 6-8  
    6.2.1.2 ServantLocator Interface ....................................... 6-9  
  6.2.2 Default Servants ......................................................... 6-10  
  6.2.3 Adapter Activators ..................................................... 6-10  
6.3 Activating Objects .......................................................... 6-12  
  6.3.1 Identifying Objects ..................................................... 6-12  
  6.3.2 Explicitly Activating Objects ......................................... 6-13  
    6.3.2.1 Using POA-assigned Object IDs .............................. 6-13  
    6.3.2.2 Application-assigned Object ID ............................. 6-14  
  6.3.3 Activating Objects on Demand ....................................... 6-15  
  6.3.4 Activating Objects Implicitly ....................................... 6-16  
  6.3.5 Activating Persistent Objects ...................................... 6-17  
6.4 Activating the POA Manager .............................................. 6-18  
6.5 Deactivating Objects ....................................................... 6-21  

Chapter 7 Portable
# Interceptors

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Introduction</td>
<td>7-1</td>
</tr>
<tr>
<td>7.2 Request Interceptors</td>
<td>7-1</td>
</tr>
<tr>
<td>7.3 ORBInitializer Object</td>
<td>7-1</td>
</tr>
<tr>
<td>7.3.1 Registering Request Interceptors</td>
<td>7-2</td>
</tr>
<tr>
<td>7.3.2 Ordering Request Interceptors</td>
<td>7-2</td>
</tr>
<tr>
<td>7.4 Naming Interceptors</td>
<td>7-3</td>
</tr>
<tr>
<td>7.5 Interception Points</td>
<td>7-3</td>
</tr>
<tr>
<td>7.6 IOR Interceptor Points</td>
<td>7-5</td>
</tr>
<tr>
<td>7.6.1 Registering IOR Interceptors</td>
<td>7-5</td>
</tr>
<tr>
<td>7.7 Invoking IOR interceptors</td>
<td>7-5</td>
</tr>
<tr>
<td>7.8 Tracing Facility Example</td>
<td>7-6</td>
</tr>
<tr>
<td>7.8.1 Registration</td>
<td>7-6</td>
</tr>
<tr>
<td>7.8.2 Initialization Class</td>
<td>7-6</td>
</tr>
<tr>
<td>7.8.3 Request Interceptor Class</td>
<td>7-7</td>
</tr>
<tr>
<td>7.8.4 Killing the Server</td>
<td>7-8</td>
</tr>
<tr>
<td>7.8.5 Server Class</td>
<td>7-8</td>
</tr>
<tr>
<td>7.8.6 Client Class</td>
<td>7-9</td>
</tr>
</tbody>
</table>

## Chapter 8 DII/DSI

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 Static Requests</td>
<td>8-1</td>
</tr>
<tr>
<td>8.2 Dynamic Requests</td>
<td>8-1</td>
</tr>
<tr>
<td>8.3 Invoking Dynamic Requests</td>
<td>8-1</td>
</tr>
<tr>
<td>8.4 Receiving Requests Dynamically</td>
<td>8-1</td>
</tr>
<tr>
<td>8.5 Dynamic Any's</td>
<td>8-1</td>
</tr>
<tr>
<td>8.6 Performance Issues</td>
<td>8-2</td>
</tr>
<tr>
<td>8.7 Demonstration Programs</td>
<td>8-2</td>
</tr>
</tbody>
</table>

## Chapter 9 RMI-IIOP

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1 Introduction to RMI-IIOP</td>
<td>9-1</td>
</tr>
<tr>
<td>9.2 Using RMI-IIOP with orb2 for Java</td>
<td>9-1</td>
</tr>
<tr>
<td>9.3 Special Codeset Requirements</td>
<td>9-1</td>
</tr>
<tr>
<td>9.4 RMI/IDL Subset</td>
<td>9-2</td>
</tr>
<tr>
<td>9.4.1 Primitive Types</td>
<td>9-2</td>
</tr>
<tr>
<td>9.4.2 RMI Remote Interfaces</td>
<td>9-2</td>
</tr>
<tr>
<td>9.4.3 RMI Value Types</td>
<td>9-2</td>
</tr>
<tr>
<td>9.4.4 Arrays</td>
<td>9-2</td>
</tr>
<tr>
<td>9.4.5 Exceptions</td>
<td>9-2</td>
</tr>
<tr>
<td>9.4.6 CORBA Object References</td>
<td>9-2</td>
</tr>
<tr>
<td>9.4.7 IDL Entity Types</td>
<td>9-2</td>
</tr>
<tr>
<td>9.5 Building an RMI-IIOP Server Program</td>
<td>9-2</td>
</tr>
<tr>
<td>9.5.1 Define the interface</td>
<td>9-2</td>
</tr>
<tr>
<td>9.5.2 Implement the interface</td>
<td>9-3</td>
</tr>
<tr>
<td>9.5.3 Generate Skeleton Classes</td>
<td>9-3</td>
</tr>
<tr>
<td>9.5.4 Implement the Server</td>
<td>9-3</td>
</tr>
<tr>
<td>9.6 Building an RMI-IIOP Client Program</td>
<td>9-4</td>
</tr>
<tr>
<td>9.6.1 Locate Object Reference</td>
<td>9-4</td>
</tr>
<tr>
<td>9.6.2 Invoking Operations</td>
<td>9-5</td>
</tr>
</tbody>
</table>
## Contents

<table>
<thead>
<tr>
<th>Number</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.7</td>
<td>RMI Codebase ..........................................................</td>
</tr>
<tr>
<td>9.8</td>
<td>CORBA Clients and Servers ............................................</td>
</tr>
<tr>
<td>9.8.1</td>
<td>CORBA-IIOP Client and RMI-IIOP Server ................................</td>
</tr>
<tr>
<td>9.8.2</td>
<td>RMI-IIOP Client and CORBA-IIOP Service ..........................</td>
</tr>
</tbody>
</table>

## Chapter 10 Class Library

<table>
<thead>
<tr>
<th>Number</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1</td>
<td>com.twoab.orb2.util.IORFormat ..................................</td>
</tr>
<tr>
<td>10.2</td>
<td>com.twoab.orb2.util.IORFormatException .........................</td>
</tr>
<tr>
<td>10.3</td>
<td>com.twoab.orb2.util.IORPrint ....................................</td>
</tr>
<tr>
<td>10.4</td>
<td>com.twoab.orb2.util.ProfileFormat ................................</td>
</tr>
<tr>
<td>10.5</td>
<td>com.twoab.orb2.util.SystemExceptionFormat .......................</td>
</tr>
<tr>
<td>10.6</td>
<td>com.twoab.orb2.util.TaggedComponentFormat .......................</td>
</tr>
<tr>
<td>10.7</td>
<td>com.twoab.orb2.util.TaggedProfileFormat .........................</td>
</tr>
<tr>
<td>10.8</td>
<td>com.twoab.orb2.Trader ...............................................</td>
</tr>
<tr>
<td>10.9</td>
<td>com.twoab.orb2.Trading .............................................</td>
</tr>
</tbody>
</table>
Figures

Figure 2.1  IDL for the Calculator Object (Calc.idl) ................................................. 2-2
Figure 2.2  Compiling the IDL .................................................................................. 2-3
Figure 2.3  Code Generated for Calculator interface .................................................. 2-4
Figure 2.4  Code Generated for CalculatorOperations.java ....................................... 2-4
Figure 2.5  Code Generated for CalculatorPOA.java .................................................. 2-5
Figure 6.1  POA Connects Servant to ORB Example .................................................. 6-1
Figure 6.2  POAs Form Hierarchical Namespace ......................................................... 6-3
Figure 6.3  POA Processing States .............................................................................. 6-20
Figures
### Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 0.1</td>
<td>OMG Documentation</td>
<td>0-x</td>
</tr>
<tr>
<td>Table 6.1</td>
<td>POA Interface Operations</td>
<td>6-2</td>
</tr>
<tr>
<td>Table 6.2</td>
<td>Default Policies for the Root POA</td>
<td>6-4</td>
</tr>
<tr>
<td>Table 6.3</td>
<td>POA Policies</td>
<td>6-5</td>
</tr>
<tr>
<td>Table 6.4</td>
<td>Servant Retention and Request Processing Policy Behaviors</td>
<td>6-7</td>
</tr>
<tr>
<td>Table 6.5</td>
<td>POA Manager States</td>
<td>6-19</td>
</tr>
<tr>
<td>Table 7.1</td>
<td>Client/Server Interception Points</td>
<td>7-3</td>
</tr>
<tr>
<td>Table 7.2</td>
<td>Starting/Intermediate/Ending Interception Points</td>
<td>7-4</td>
</tr>
</tbody>
</table>
About This Document

What is orb2 for Java?

The orb2 for Java product provides an Object Request Broker (ORB) that is compliant with the Common Object Request Broker Architecture (CORBA) Specification (Version 3.0). It does not support the messaging APIs described in Chapter 22 of the specification.

It supports the Java language bindings that are compliant with the IDL to Java Language Mapping Specification (Version 1.2).

It supports the Java RMI-IIOP protocol that is compliant with the Java to IDL Language Mapping Specification (Version 1.2).

It supports the Naming Service that is compliant with the Naming Service Specification (Version 1.2).

The orb2 for Java product supports the following features:

• CORBA IDL support
• CORBA Java language mapping
• RMI-IIOP
• CSIv2 Security
• Portable Object Adapter
• Portable Interceptors
• Multi-threading
• Dynamic Invocation Interface
• Dynamic Skeleton Interface
• Dynamic Any
• Naming and Trader Services
• eXplorer Console
• Simple configuration
• Easily embedded

What does this guide include?

This guide provides information on configuring and running the orb2 for Java environment. It also provides some elementary documentation for CORBA developers. It does not purport to provide complete documentation of the CORBA specifications. These documents are freely available and should be understood by those that develop systems using this product. These documents are referenced in Section, OMG Documentation, below.
OMG Documentation

Table 0.1 lists documentation provided by the Object Management Group (OMG)

Table 0.1 OMG Documentation

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA/IIOP</td>
<td>formal/2002-11-01</td>
<td>3.0.1</td>
</tr>
<tr>
<td>IDL to Java</td>
<td>formal/2002-08-05</td>
<td>1.2</td>
</tr>
<tr>
<td>Java to IDL</td>
<td>formal/2002-08-06</td>
<td>1.2</td>
</tr>
<tr>
<td>Naming Service</td>
<td>formal/2002-09-02</td>
<td>1.2</td>
</tr>
<tr>
<td>Security Service</td>
<td>formal/2002-03-11</td>
<td>1.8</td>
</tr>
</tbody>
</table>

Technical Support

Your feedback is important to us. Please provide input to the 2AB Technical Support Staff.
Supported customers may communicate comments to and request help from the Technical Support Staff via the following methods:

Email
support@2ab.com

Telephone
+ 1.205.621.7455
Chapter 1  Getting Started

This chapter will provide the information required to install, configure and run example systems that utilize the orb2 for Java product.

1.1 CORBA Interoperability

The orb2 for Java product supports the OMG’s IIOP protocol. It supports versions 1.0, 1.1 and 1.2 and should interoperate with any ORB product that supports this protocol. This includes the orb2 for C, C++ and Eiffel products. The orb2 for Java product also supports the OMG’s RMI-IIOP protocol for interoperating with Java applications that use RMI-IIOP (e.g. J2EE Application Servers).

1.2 Development Environment

Applications that are developed to use the orb2 for Java product should use the Java Development Environment (JDK) 1.6 or higher. Some functionality will require the use of JDK 1.6 or higher (e.g. RMI-IIOP POA-based servers). These special requirements will be documented with the specific functionality.

1.3 Runtime Environment

Applications that run using orb2 for Java product should use the Java Runtime Environment (JRE) 1.6 or higher.

1.4 Licensing

The orb2 for Java product requires a license file that will be provided by 2AB when the product is purchased. The license file will have permissions for runtime environments and/or development environments. The file must be placed in the lib subdirectory of the orb2 for Java installation directory. For runtime environments, the application may specify where the license will be located. There may be at most one license file located in the specified directory. An application can specify the directory where the license is located by either using the com.twoab.orb2.license.dir ORB property or by using the -ORBlicense_dir command-line argument.

This file must be available on each machine where an instance of the ORB will run. This includes both client and server applications. For SDK environments, a license file must be located in the lib subdirectory of the orb2 for Java installation. This is required to run the idlc compiler program and the demonstration programs.

1.5 Environmental Variables

There is no requirement to set any environmental variables when running applications using the orb2 for Java product; however, there are several scripts/applications located in the bin and demos subdirectories that require the setting of environmental variables. There are two environment variables that must be set when running these programs. They are the PATH and ORB2 variables. In addition, the ORB2C environmental variable must be used if the orb2 Trader is executed. The paragraphs below describe the use of each of these variables. When getting started with orb2 for Java, it is best to set these variables.
The ORB2 environment variable should be set to the path of the installation directory.
The system’s PATH environmental variable should include the bin subdirectory of the orb2 installation.
If running the orb2 Trader program, the ORB2C environmental variable must be set to the path of the installation directory.

1.6 Classpath and System Properties

When compiling applications that use orb2 for Java, the orb2.jar file (found in the installation’s lib subdirectory) must be included in the Java classpath.

When executing applications that use orb2 for Java, the orb2.jar file (found in the installation’s lib subdirectory), must be included in the Java classpath. In addition, the Java system properties org.omg.CORBA.ORBClass and org.omg.CORBA.ORBSingletonClass must be set to com.twoab.orb2.core.ORBImpl and com.twoab.orb2.core.ORBSingleton respectively. This can be done with either by using -D command-line argument or programmatically using the System.setProperty method. The above system properties can optionally be passed as parameters to the ORB::init method.

There may be additional classpath and system properties requirements for some features (e.g. CSIv2 security). Please see the additional requirements in the chapter defining the feature.

1.7 Locating Objects

A client application that needs to communicate with a distributed object must first obtain an object reference for the target object before it can invoke operations upon that object. This chapter discusses three techniques that can be used to obtain object references for distributed objects. They are getting stringified object references, using a Naming Service and using a Trader Service. There are other techniques that could be used, such as objects providing references for other objects in the same service, but these other techniques will not be discussed in this chapter.

1.7.1 Stringified Object References

A service that creates an object reference may create a textual representation of that object reference that is referred to as a stringified object reference. Once a stringified object reference has been created, it can be communicated to clients through a variety of means. It may be file transferred to another machine, e-mailed or it might even be stored in a well-known place on a file system.

Using an ORB’s object_to_string method creates stringified object references.

Example

```
org.omg.CORBA.Object obj = ???  // object reference created in some manner
String ior = orb.object_tostring(obj);  // create the stringified object reference
```

Clients that want to use stringified object references must have prior knowledge of where the stringified object reference can be obtained (e.g. from a specified file). Once the stringified object reference has been obtained, it can create the object reference with the ORB’s string_to_object method. This would be done as follows::

```
org.omg.CORBA.Object obj = orb.string_to_object(ior_string);
```
1.7.2 Naming Service

The OMG provides a specification for a Naming Service that allows object references to be stored in a hierarchical name tree. Services may store object references using a Naming Service, and client applications may retrieve object reference if they understand the naming scheme used by the service. A complete description on the use of the OMG's Naming Service can be found in the OMG's Naming Service Specification. This document is retrievable as the file "formal/2002-09-02."

The orb2 product provides an implementation of the OMG's Naming Service. This service is built using Java and can be run with a Java 1.3 or higher interpreter. By default, the Naming Service uses TCP/IP port 9999 and stores its persistent data in the naming.db file located in the installation’s lib subdirectory.

Complete information required to configure, run and develop clients that use the Naming Service can be found in the orb2 Services Guide.

1.7.3 Trader Service

The Trader Service provides the means for a client to retrieve one or more object references that match both a specified service type and some arbitrary set of constraints.

Complete information required to configure, run and develop clients that use the Trader Service can be found in the orb2 Services Guide.

1.8 Demonstration Programs

The orb2 for Java product provides demonstration programs that provide examples of using some aspect of its functionality. These demonstration programs are located in the demos subdirectory of the orb2 for Java installation directory.

Each demonstration program has a Readme file that provides instructions for building and running the demo. For those demonstration programs that have been pre-built, it is only necessary to rebuild the programs if the source code is changed.
Chapter 2

Developing a Distributed orb2 Application

This chapter will discuss the steps required to build a distributed client-server system using orb2 for Java. This will be demonstrated by constructing a Calculator server that performs arithmetic calculations and returns results to client applications. Three versions of this implementation are provided in the demos subdirectory. These three demo systems are identical with the exception of how object references are located. The Simple demo stores references in a file, the Trading demo stores reference in the Trader Service and the Naming demo stores references in the Naming Service.

The principals of building distributed applications that take advantage of the orb2 for Java facilities are also introduced. It uses the code in the demonstration programs to show you how to:

- Describe object interfaces using the Interface Definition Language (IDL).
- Compile the IDL using the orb2 for Java IDL compiler to produce stub and skeleton code.
- Develop servant programs that provide the object implementations.
- Develop a client application that uses the servant program.
- Use different techniques to locate objects.
- Compile the Java programs.
- Run the distributed application.

2.1 Interfaces

You use IDL to describe the interface for each component in a CORBA environment. The interface describes how a user of the component, the client, communicates with the server component...not how the component is implemented.

The IDL for the calculator, which you can find in the Calc.idl file, is shown in Figure 2.1. It defines a module, Demos, that has one interface, Calculator, and defines one exception, DivideByZero. The Calculator interface defines five methods or operations. They are add,
subtract, multiply, divide and kill. Notice that the divide operation raises the DivideByZero exception if division by zero is attempted. Also notice that the kill operation is marked as being one-way, which means that client applications do not wait for a response.

```idl
module Demos {
    exception DivideByZero { string reason; }
    interface Calculator {
        // Operation declarations:
        long add(in long op1, in long op2);
        long subtract(in long op1, in long op2);
        long multiply(in long op1, in long op2);
        long divide(in long op1, in long op2, out long res_1)
            raises (DivideByZero);
        // kill myself:
        oneway void kill();
    }
};
```

Figure 2.1 IDL for the Calculator Object (Calc.idl)

When you compile this IDL file using the orb2 IDL compiler (idlc), it produces Java class and interface files that are referred to as stubs and skeletons. Stubs are files used by the client application. Skeletons are files used by the server application.

2.2 Compiling IDL

orb2 for Java includes an IDL compiler that produces pure Java code based on the CORBA IDL-to-Java mapping specification. As illustrated in Figure 2.2, when you execute the orb2 IDL compiler, it generates two sets of Java files, stub code for clients and skeleton code for the server.

- The stub code creates proxy objects that a client can use to invoke object reference of the interface types defined in the IDL file.
- The skeleton code provides access to objects that support those interfaces.
The orb2 IDL compiler is called `idlc`. To compile the IDL file, `Calc.idl`, issue the following command:

```
idlc Calc.idl
```

By default, all files are generated in a subdirectory, called `generated`, of the current working directory. You may, however, redirect the output using one of the compiler options. The complete syntax and options for using the IDL compiler are described in Chapter 3, Compiling IDL.

When you run `idlc`, the compiler processes the `Calc.idl` file and creates a new directory that contains the Java files. The directory created is named after the IDL module name, which is `Demos` in this example. The `Demos` subdirectory will contain the following files:

- _CalculatorStub.java
- Calculator.java
- CalculatorHelper.java
- CalculatorHolder.java
- CalculatorOperations.java
- CalculatorPOA.java
- CalculatorPOATie.java
- DivideByZero.java
- DivideByZeroHelper.java
- DivideByZeroHolder.java

Figure 2.2 Compiling the IDL
The Calculator IDL interface is mapped to a Java interface of the same name (Calculator.java). As the generated code in Figure 2.3 illustrates, the Calculator interface extends a base class for a CORBA object.

```java
package Demos;
//
// Interface definition : Calculator
//
// @author orb2 Compiler
//
public interface Calculator extends CalculatorOperations, org.omg.CORBA.Object,
    org.omg.CORBA.portable.IDLEntity
{

}
```

**Figure 2.3 Code Generated for Calculator interface**

The code generated in the CalculatorOperations.java file defines the operations supported by the interface.

```java
package Demos;
//
// Interface definition : Calculator
//
// @author orb2 Compiler
//
public interface CalculatorOperations
{
    // // Operation add
    //
    public int add(int op1, int op2);

    // // Operation subtract
    //
    public int subtract(int op1, int op2);

    // // Operation multiply
    //
    public int multiply(int op1, int op2);

    // // Operation divide
    //
    public int divide(int op1, int op2, org.omg.CORBA.IntHolder res_1)
    throws Demos.DivideByZero;

    // // Operation kill
    //
    public void kill();

}
```

**Figure 2.4 Code Generated for CalculatorOperations.java**

The Demos package contains the skeleton code for supporting the Portable Object Adapter (POA) interface (see Chapter 6, Portable Object Adapter). The classes generated from the application-supplied implementation class support both inheritance-based and delegation-based (also known as the Tie method) relationships between the object and servant. If the object implementation uses inheritance, CalculatorPOA.java obtains the implementation object. If the object implementation uses delegation, CalculatorPOATie.java obtains a reference to the implementation object.
As shown in `CalculatorPOA.java`, the PortableServer module for the POA defines the native Servant type. In Java, the Servant type is mapped to the Java `org.omg.PortableServer.Servant` class.

```java
package Demos;

// Interface definition : Calculator
//
// @author orb2 Compiler
//
public abstract class CalculatorPOA extends org.omg.PortableServer.Servant
    implements CalculatorOperations, org.omg.CORBA.portable.InvokeHandler

Figure 2.5 Code Generated for CalculatorPOA.java
```

The Servant class is a Java abstract class that serves as the base class for all POA servant implementations. It provides several methods that an application may invoke as well as methods that the POA invokes, which the application can override to control aspects of servant behavior.

You must associate the servant with an ORB instance before any other methods, except `_all_interfaces()` and `_this_object()`, are invoked or a `CORBA::BAD_INV_ORDER` exception is raised.

### 2.3 Developing Server Programs

For each server object, a servant must exist that implements the operations of the CORBA object. The servant provides the implementation of the operations described for the CORBA object interface.

You write an implementation class to define an implementation in Java. Instances of the implementation class implement the IDL interfaces. The implementation class must define public methods that correspond to the operations and attributes of the IDL interface supported by the object implementation (as defined by the mapping specification for IDL interfaces). Providing these methods satisfies the abstract methods defined by a specific interface skeleton class.

The association between a CORBA object and a servant does not need to be one-to-one. A servant can provide the implementation for many CORBA objects; a CORBA object can be implemented by many servants in its lifetime. A servant is connected to the ORB via the POA. A POA called the `RootPOA` is always available to applications and is used in the Calculator demonstrations.

The Servant class is a Java abstract class that serves as the base class for all POA servant implementations. It provides a number of methods that you can invoke, as well as methods that the POA invokes that you can override, to control aspects of servant behavior.

All methods defined in the Servant class, except the `_all_interfaces()` and `_this_object(ORB orb)` methods, are invoked only after the servant is associated with an ORB instance. If the servant was not associated with an ORB instance when invocation of any other methods are attempted, raises an exception, `CORBA::BAD_INV_ORDER`.

You can associate a servant with an ORB instance using one of the following:

- By calling `_this_object(ORB orb)` and passing an ORB instance as a parameter to associate the servant with the specified ORB instance.
- By explicitly activating a servant with a POA by calling either `POA::activate_object` or `POA::activate_object_with_id` to associate the servant with the ORB instance that contains the POA on which the servant was activated.
• By returning a servant instance from a ServantManager. The servant returned from
PortableServer::ServantActivator::incarnate() or
PortableServer::ServantLocator::preinvoke() is associated with the ORB instance that
contains the POA on which the ServantManager is installed.
• By installing the servant as a default servant on a POA. The servant is associated with the
ORB instance that contains the POA for which the Servant is acting as a default servant.
• By explicitly calling org.omg.CORBA_2_3.ORB.set_delegate().

The Calculator demonstration programs use the _this_object() method to associate its servant
with an instance of the orb2 ORB. Chapter 6, Portable Object Adapter, describes how to use
ServantManagers and default servants.

2.3.1 Developing the Calculator Server Program

The Server.java file contains the server startup code (in the main operation) and the
CalculatorServant class that inherits from the skeleton class. This section describes the
significant code in the example.

1. Declare a class called Server and begin the main program.

```java
public class Server {
    public static void main(String[] args) {
```

2. Initialize properties to specify orb2 for Java.

```java
Properties props = new Properties();
props.setProperty("org.omg.CORBA.ORBClass",
    "com.twoab.orb2.core.ORBImpl");
props.setProperty("org.omg.CORBA.ORBSingletonClass",
    "com.twoab.orb2.core.ORBSingleton");
props.setProperty("com.twoab.orb2.license.dir",
    System.getProperty("user.dir"));
```

3. This code initializes an instance of the ORB. The init() function is a static member function
in the ORB class. It initializes the ORB and returns an ORB instance. The first parameter
of init() is a list of strings, which provide the command-line arguments for initializing the
ORB. The second parameter, if it is not null, is a list of ORB properties. Refer to
CORBA::ORB in Chapter 4, Initializing the ORB, for a complete list of the initialization
parameters.

```java
try {
    orb = ORB.init(args, props);
```

4. Obtain an object reference to the Root POA. The ORB operation,
resolve_initial_references(), allows CORBA applications to acquire object references at
process startup. A string is passed representing the service required. the Root POA is
always available via the resolve_initial_references() operation, which returns a CORBA
object of type CORBA.Object. The narrow operation on the helper classes narrow the type
of object returned from resolve_initial_references. A POAHelper.narrow operation
narrow the object parameter to an org.omg.PortableServer.POA type.

```java
org.omg.CORBA.Object obj = orb.resolve_initial_references("RootPOA");
root_poa = POAHelper.narrow(obj);
```
5. Associate the Calculator servant with an ORB instance before implicitly activating it in the next step. Although there are several ways to associate a servant with an ORB instance, the demonstration programs call `_this_object(orb)` in the `CalculatorServant` constructor.

```java
CalculatorServant calc = new CalculatorServant(orb, poa);
```

6. Implicitly activate the Calculator servant with the Root POA.

```java
Demos.Calculator calc_ref = calc._this();
```

7. Activate the POAManager so that registered POAs are ready to receive requests. Note All POAManagers are created in a holding state; any incoming requests are held in a queue. You must change the state of the POAManager associated with the Root POA from the holding state to the active state to allow the dispatching of requests. Otherwise, incoming requests are queued. See Section 6.4, Activating the POA Manager.

```java
poa.the_POAManager().activate();
```

8. Save the object reference so that client applications can locate the component. The Simple demonstration saves the reference in a file as follows.

```java
String str_ref = orb.object_to_string(calc_ref);
File file = null;
FileWriter out = null;
file = new File("server.ior");
out = new FileWriter(file);
int size = str_ref.length();
int written = 0;
while(written < size - 80) {
    out.write(str_ref, written, 80);
    written += 80;
}
out.write(str_ref, written, size - written);
out.close();
```

9. The Trading demonstration must first locate the object reference for the Trader Service and register the object reference with the Trader Service.

```java
com.twoab.orb2.Trading trading =
    com.twoab.orb2.TradingHelper.narrow(
        orb.resolve_initial_references("orb2Trading"));
trading.register("Calculator", "/", ",", calc_ref);
```
10. The Naming demonstration must first locate the object reference for the Naming Service and register the object reference with the Naming Service.

```java
org.omg.CosNaming.NamingContext root_ctx = 
    org.omg.CosNaming.NamingContextHelper.narrow ( 
        orb.resolve_initial_references("NameService"));

org.omg.CosNaming.NameComponent nc =
    new org.omg.CosNaming.NameComponent("demos", "naming");
org.omg.CosNaming.NameComponent[] name = {nc};
try {
    root_ctx.resolve(name);
    root_ctx.bind_new_context(name);
    // this is ok;
} catch (org.omg.CosNaming.NamingContextPackage.AlreadyBound ab) {
    // this is ok;
} catch (Exception e) {
    System.out.println("Exception resolving name/binding new context");
    System.exit(1);
} 
root_ctx.rebind(name, calc_ref);
```

11. The ORB listens for requests. The run function waits for requests; the function returns only after `ORB.shutdown` is called.

```java
orb.run();
}
catch( Exception e) {
    e.printStackTrace();
    System.err.println("Exception occurred :: "+e.toString());
} 
```
12. The Calculator implementation code implements the operations specified in the 
   Calculator interface with the CalculatorServant class inheriting from the 
   Demos.CalculatorPOA class generated by the orb2 IDL compiler. You associate this 
   servant with an ORB instance by calling _this_object().

   ```java
   class CalculatorServant extends Demos.CalculatorPOA {
       private ORB orb;
       private POA root_poa;

       /**
        * Constructs instance of the servant.
        */
       CalculatorServant(ORB orb, POA root_poa) {
           _this_object(orb);
           this.orb = orb;
           this.root_poa = root_poa;
       }

       /**
        * Implements add operations.
        */
       public int add(int op1, int op2) {
           //Refer to the Server.java file for the remainder
           //of the code for the arithmetic operations
       }
   }
   ```

   An alternative method of connecting the servant implementation class to the ORB is to use 
delegation instead of inheritance. This is useful because Java only allows single inheritance of 
implementations; that is, classes. For example, if a legacy servant is already inheriting from its 
own implementation class, it cannot inherit from these compiler-generated classes. In this case, 
use the delegation method to connect this servant to the POA.

   One of the classes generated by the IDL compiler is a Tie class. The Tie class inherits from the 
interface-namePOA skeleton class and includes implementations of all the operations defined 
in the IDL interface. The implementations of the IDL interface operations invoke the same 
operation on a delegate object. The delegate object is set when the Tie class is instantiated. It is 
the delegate object that provides the full implementations of the operations. When an upcall is 
received, the Tie class operation is called and the Tie class then reinvokes the same operation 
on the delegate servant.

   For Tie-based servants, the Tie template class overrides the virtual method it inherits from 
servant and from the type-specific POA skeleton base classes. The instance of the Tie template 
class, not the tied object, is registered with the POA as the servant. Ties allow classes that are 
not related to skeletons by inheritance to implement object operations. Typically, there is no 
inheritance relationship to the IDL-generated skeleton classes. This makes this approach easier 
to integrate legacy code. For each operation they supply, the Tied object must have a method 
with exactly the same signature.

### 2.4 Developing Client Programs

Developing client applications is simply a matter of obtaining an object reference for some 
distributed component and then using that object reference to invoke remote operations. The 
manner in which one locates an object reference depends on the manner in which the server 
programs make them available. The Simple, Trading and Naming demonstration programs 
illustrate three common methods of making the object reference of a service available to client 
application. They are:

- Write stringified copies of the object reference to a file.
• Use the Trader Service.
• Use the Naming Service.

### 2.4.1 Developing the Calculator Client Program

The demonstration Calculator clients (Simple, Naming and Trading) run as command-line applications. The client application accepts command-line parameters for the arithmetic calculation to be performed and passes these parameters to the Calculator server. When the Calculator server returns the result, the client application displays the result on the console window.

A CORBA client application performs these basic steps:

- Initializes the orb2 ORB.
- Obtains the object reference for the object on which it wants to invoke operations.
- Invokes the operations and processes the results.
- Releases the object reference and shuts down the ORB.

The following are excerpts from the client code for the Calculator console. Refer to the `Client.java` files in the `demos/simple`, `demos/trading` and `demos/naming` directories for the complete program.

1. Initialize the ORB by calling the `init()` method on the `org.omg.CORBA.ORB` class.

```java
try {
    // initialize ORB
    orb = ORB.init(args, props);
}
```

2. You can use several techniques to locate object references, the Simple demonstration obtains the object reference from a file.

```java
File file = new File("server.ior");
if(!file.exists() || !file.isFile() || !file.canRead()) {
    System.out.println("Cannot read server.ior file.");
    System.exit(1);
}
FileReader in = new FileReader(file);
int size = (int)file.length();
char [] data = new char[size];
int chars_read = 0;
while(chars_read < size) {
    chars_read += in.read(data, chars_read, size - chars_read);
}
in.close();
// create object reference
org.omg.CORBA.Object obj = orb.string_to_object(new String(data));
Demos.Calculator calc = Demos.CalculatorHelper.narrow(obj);
```
3. In the Trading demonstration, the object reference is obtained from the Trader Service.

```java
// locate the Calculator server via the Trader:
com.twoab.orb2.Trading trading =
    com.twoab.orb2.TradingHelper.narrow(
        orb.resolve_initial_references("orb2Trading"));
com.twoab.orb2.TraderPackage.OffersHolder offers =
    new com.twoab.orb2.TraderPackage.OffersHolder();
org.omg.CORBA.Object obj = trading._import("Calculator", "/", "/");
Demos.Calculator calc = Demos.CalculatorHelper.narrow(obj);
```

4. In the Naming demonstration, the object reference is obtained from the Naming Service.

```java
// Initialize object reference for Naming Service
org.omg.CosNaming.NamingContext root_ctx =
org.omg.CosNaming.NameComponent nc =
    new org.omg.CosNaming.NameComponent("demos", "naming");
org.omg.CosNaming.NameComponent[] name = {nc};
org.omg.CORBA.Object obj = root_ctx.resolve(name);
Demos.Calculator calc = Demos.CalculatorHelper.narrow(obj);
```

5. Perform the requested calculator operations using the servant. You can invoke the methods using the object reference returned by the Trader. The servant performs the calculation and the result is returned to the client. The client prints to the standard output. System and user exceptions that are raised are caught and processed.

```java
try {
    switch( op) {
        case '+':
            int sum = calc.add(left_operand, right_operand);
            System.out.println(
                "Result: " + left_operand + " + " + right_operand + " = " + sum);
            break;
    //Refer to the Client.java file for the remainder
    //of the client calculator operations code.
}
```

6. When processing is complete, the application needs to release the Calculator object reference and shuts down the ORB.

```java
calc._release();
orb.shutdown (true);
```
Chapter 3  Compiling IDL

3.1 Introduction

CORBA-based services implement interfaces that are specified by the Interface Definition Language (IDL). The syntax and semantics of IDL are described completely in the OMG CORBA/IIOP specification (Section, OMG Documentation).

An IDL compiler is used to interpret the IDL that defines a service and generates stub and skeleton class files that are used by both the service and its client applications. Client applications use the generated stub classes. Services use the generated skeleton classes. These generated classes and their use are defined in the IDL to Java Language Mapping specification (Section, OMG Documentation).

This release of orb2 for Java provides the idlc compiler. Instructions for running the idlc compiler are found in the following section.

3.2 The idlc IDL Compiler

The idlc compiler is a fully functional IDL compiler that generates IDL stub and skeleton classes from IDL files that conforms to the OMG's CORBA standard for IDL. The idlc compiler is based on a modified version of the IDL compiler that is packaged with the OpenOrb Object Request Broker. The following section provides the copyright notice that allows the modification and distribution of this software.

3.2.1 Running the idlc Compiler

The idlc compiler is started from a script located in the bin subdirectory of the orb2 for Java installation. This directory should be included in the path specified by your PATH environment variable.

The compiler is run at a command-line prompt as follows:

```
idl -option <option> ... idl_file_name
```

where `<option>` is a command-line option (their may be multiple options used) and `idl_file_name` is the name if the IDL file to be compiled.

The following sections define the supported command-line options.

3.2.1.1 General Options

- `-help`
  Displays command line options. The argument "-h" does the same thing. Running the compiler with no arguments does the same thing.

- `-release`
  Displays the current release number for the idlc compiler.

- `-directory_name`
  This will cause the generated stub and skeleton classes to be placed under the directory specified by `directory_name`. If this option is not specified, output will be placed in a directory named `generated` that is a subdirectory of the current directory.
-I directory_name

This will cause the compiler to search for include files in the directory named directory_name.

-D a_symbol

This specifies that the symbol a_symbol is defined as though it had been defined with a #define statement in an IDL file.

### 3.2.1.2 Package Name Options

Java package names for idlc generated stub/skeleton files consists of two parts. The first part we call the **package prefix**. The second part is the **module name**. For IDL interfaces and data types that are not defined within an IDL module, the module name will be a zero length string.

The default behavior for the idlc compiler when no options are specified is to generate classes with a **package prefix** that is derived from the #pragma prefix name(s) contained in the IDL file. The **package prefix** is the reverse of the prefix name specified with the #pragma prefix statement(s). For example, #pragma prefix twoab.com, the **package prefix** will be com.twoab.

The **module name** is derived from the IDL module name. Appending the **module name** to the package prefix forms the Java package name. For example, #pragma prefix twoab.com with module A will cause classes to be generated into a com.twoab.A Java package.

The idlc compiler supports several options that may be used to modify this default Java package naming convention and/or to specify package names of included IDL that does not follow this naming convention. The options described below allow full control of the Java package names for generated stubs and skeletons.

- noprefix

This causes the #pragma prefix name(s) contained in the IDL to NOT be used to create the **package prefix**. This option may be used in conjunction with the –package option (defined below) which can be used to fully control the **package prefix**.

- noreverseprefix

This causes the #pragma prefix to be used in the **package prefix** without reversing. This argument is provided to correct older versions of IDL files that specified the #pragma prefix incorrectly. For example, #pragma prefix com.twoab with a “–noreverseprefix” command line argument will cause classes to be generated into a com.twoab package prefix if no other command line options are specified. This option may be used in conjunction with the –package option (defined below) to fully control the **package prefix**.

- package package_name

This causes the **package prefix** to have package_name prepended to its value. This option works in conjunction with the default behavior and/or any other command line options controlling packages (defined above) to fully control the **package prefix**.

Examples:

1) IDL file: #pragma prefix "twoab.com"
   
   idlc options: –package utility
   
   The package prefix will be utility.com.twoab
   
   The Java package name for module A will be utility.com.twoab.A

2) IDL file: #pragma prefix "twoab.com"
   
   idlc options: –package utility -noprefix
The package prefix will be *utility*

The Java package name for module A will be *utility.A*

3) IDL file: 

    #pragma prefix "com.twoab"

**idlc options:**

- *–package utility -noreverseprefix*

The package prefix will be *utility.com.twoab*

The Java package name for module A will be *utility.com.twoab.A*

**-include_pkg module_name=package_name**

This option (or the *–include_pkg_file* option) is necessary when *idlc* options that modify the default behavior of the Java package name have been used to compile IDL that is included in this IDL file (-package, -noprefix, and/or -noreverseprefix). It may also be necessary if third-party provided stubs/skeletons are used which did not follow the reverse prefix naming convention for Java packages when compiled.

This option causes references to modules contained in an included IDL file to use the package prefix *package_name* for the module *module_name* when generating stubs and skeletons. This *package_name* is the fully qualified package prefix.

For example:

IDL file1.idl has:

    #pragma prefix twoab.com

The *idlc* options used to compile: 

- *–package utility*

The Java package name for module A will be *utility.com.twoab.A*

IDL file2.idl has:

    #pragma prefix twoab.com

    #include file1.idl

    … some reference to module A from file1.idl…

The *idlc* options used: 

- *-include_pkg A=utility.com.twoab*

The package prefix is com.twoab for classes generated by this compile (no overrides to default), however references to module A in these stubs/skeletons will be prefixed with utility.com.twoab

**-include_pkg_file file_name**

This option (or the *–include_pkg* option) is necessary when *idlc* options (-package, -noprefix, and/or -noreverseprefix) that modify the default behavior of the Java package name have been used to compile IDL that is included in this IDL file. It may also be necessary if third-party provided stubs/skeletons are used which did not follow the reverse prefix naming convention of the *idlc* compiler for Java packages.

This option allows a text file to be used to provide the names of the included modules and their associated package prefixes. The functionality is equivalent to using multiple *–include_pkg* options on the command line.

Each line in the file has the format:

```
module_name package=package_prefix
```

where *module_name* is the name of a module in an included file and *package_name* specifies the package prefix for the module. Text file lines that begin with // or # will be ignored. The *package=package_prefix* token must not have embedded spaces. Additional tokens may exist on the line (e.g. prefix=xxx); however, they will be ignored.
This file entry causes references to modules contained in an included IDL file to use the package prefix named `package_prefix` for the module `module_name` when generating stubs and skeletons.

Note: If the `-include_pkg` option is also specified for a module, it will override any entries for that module specified in the file.

### 3.2.1.3 Stub/Skeleton Generation Options

- **-all**
  
  This will cause stub and skeleton classes to be generated for all included IDL files. The stub and skeleton classes will be in Java packages based on the options specified for the creation of the primary IDL file to be compiled. That is, all command line options will be treated as global and each include IDL file will be compiled using the rules described in Section 4.2.1.2.

- **-noskeleton**
  
  This causes skeleton classes to not be generated.

- **-nostub**
  
  This causes stub classes to not be generated.

- **-notie**
  
  This causes the generation of TIE classes to not be generated.

### 3.2.2 Running Without the idlc Script

The `idlc` compiler is a program written using the Java programming language. It is typically started using the `idlc` script; however, it may be started directly with the Java interpreter. The class that is run is `com.twoab.orb2.compiler.IdlCompiler`. The command-line arguments are identical to those accepted by the `idlc` script. This class and all other classes required to run the compiler are found in the `orb2compiler.jar` file.

### 3.2.3 OpenOrb Copyright

Redistribution and use of this software and associated documentation ("Software"), with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain copyright statements and notices. Redistributions must also contain a copy of this document.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The name "Exolab" must not be used to endorse or promote products derived from this Software without prior written permission of Intalio Inc. For written permission, please contact info@exolab.org.

4. Products derived from this Software may not be called "Exolab" nor may "Exolab" appear in their names without prior written permission of Intalio Inc. Exolab is a registered trademark of Intalio Inc.

5. Due credit should be given to the Exolab Project (http://www.exolab.org/).

THIS SOFTWARE IS PROVIDED BY INTALIO AND CONTRIBUTORS `AS IS` ANDANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL INTALIO OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright 2000 (C) Intalio Inc. All Rights Reserved.

3.3 The idl2wsdl IDL Compiler

The idl2wsdl compiler is a fully functional IDL compiler that generates Web Services Definition Language (WSDL) from IDL files that conforms to the OMG's CORBA standard for IDL to WSDL. The idl2wsdl compiler uses the IDL parser that is part of the idlc compiler and is based on a modified version of the IDL compiler that is packaged with the OpenOrb Object Request Broker. The previous section provides the copyright notice that allows the modification and distribution of this software. The wsdl and brige generation code which is used is not a derivation of any other source.

3.3.1 Running the idl2wsdl Compiler

The idl2wsdl compiler is started from a script located in the bin subdirectory of the orb2 for Java installation. This directory should be included in the path specified by your PATH environment variable.

The compiler is run at a command-line prompt as follows:

```
idl2wsdl <option> <option> .... idl_file_name
```

where `<option>` is a command-line option (there may be multiple options used) and `idl_file_name` is the name if the IDL file to be compiled.

The following sections define the supported command-line options.

3.3.1.1 General Options

- **-help**
  Displays command line options. The argument "-h" does the same thing. Running the compiler with no arguments does the same thing.

- **-release**
  Displays the current release number for the idl2wsdl compiler.

- **-d directory_name**
  This will cause the generated stub and skeleton classes to be placed under the directory specified by `directory_name`. If this option is not specified, output will be placed in a directory named `generated` that is a subdirectory of the current directory.

- **-I directory_name**
  This will cause the compiler to search for include files in the directory named `directory_name`.

- **-D a_symbol**
  This specifies that the symbol `a_symbol` is defined as though it had been defined with a `#define` statement in an IDL file.
3.3.1.2 Package Name Options

Java package names for idl2wsdl generated files consists of two parts. The first part we call the **package prefix**. The second part is the **module name**. For IDL interfaces and data types that are not defined within an IDL module, the module name will be a zero length string.

The default behavior for the idl2wsdl compiler when no options are specified is to generate classes with a **package prefix** that is derived from the #pragma prefix name(s) contained in the IDL file. The **package prefix** is the reverse of the prefix name specified with the #pragma prefix statement(s). For example, `#pragma prefix twoab.com`, the **package prefix** will be `com.twoab`.

The **module name** is derived from the IDL module name. Appending the **module name** to the package prefix ultimately forms the Java package name. For example, if no command line options are specified, `#pragma prefix twoab.com` with module `A` will cause classes to be generated into a `com.twoab.A` Java package.

The idl2wsdl compiler supports several options that may be used to modify the default package naming convention and/or to specify package names of included IDL that does not follow this naming convention. The options described below allow full control of the package names.

**Note** You should use the same options here that you use on the idl2c compile of your idl if you are planning to build a corba to soap bridge for this service. See Section 3.2.1.2, Package Name Options for a full definition of these options.

### 3.3.1.3 WSDL options

- **-tns target_name_space**
  - The target name space for the WSDL. The default is: http://default_namespace

- **-soap_addr soap_address**
  - The soap address. The default is: soap_address

- **-rpc true_false**
  - Determine whether or not RPC style bindings will be generated

### 3.3.2 Running Without the idl2wsdl Script

The idl2wsdl compiler is a program written using the Java programming language. It is typically started using the idl2wsdl script; however, it may be started directly with the Java interpreter. The class that is run is `com.twoab.orb2.compiler.Idl2Wsdl`. The command-line arguments are identical to those accepted by the idl2wsdl script. This class and all other classes required to run the compiler are found in the `orb2compiler.jar` file.
Chapter 4 Initializing the ORB

Both client and server applications that utilize CORBA must initialize an ORB within their programs. The parameters that are used to initialize an ORB are obtained from both command-line arguments and predefined properties. A property is a name-value pair that defines the property name and its value. These initialization parameters can be obtained from command-line arguments, special configuration files and Java System properties. This chapter will define the command-line parameters and properties that can be used to initialize an orb2 ORB.

4.1 Writing the Initialization Software

orb2 can be initialized in the following manner:

```java
import org.omg.CORBA.*;

public static void main(String[] args) {
    java.util.Properties properties = System.getProperties();
    properties.setProperty("org.omg.CORBA.ORBClass", "com.twoab.orb2.core.ORBImpl");
    properties.setProperty("org.omg.CORBA.ORBSingletonClass", "com.twoab.orb2.core.ORBSingleton");
    props.setProperty("com.twoab.orb2.license.dir", System.getProperty("user.dir"));
    ORB orb = ORB.init(args, properties);
}
```

In the above example, the properties are specifying that the orb2 ORB should be used rather than the ORB provided with the JDK. This could have also be specified with the -D option of the Java interpreter. The application would then be started as follows:

```bash
java -Dorg.omg.CORBA.ORBClass=com.twoab.orb2.core.ORBImpl -Dorg.omg.CORBA.ORBSingletonClass=com.twoab.orb2.core.ORBSingleton TheApplication
```

4.2 ORB Property Files

In addition to the properties that an application might pass to the ORB.init method, the ORB might also obtain properties to be used from special files called property files. A property file is a text file that contains one or more name-value pairs each representing a property name and value. An ORB property file would look something like the following:

```plaintext
com.twoab.orb2.port=9002
com.twoab.orb2.host=hq_server
com.twoab.orb2.jthreads.timeout=60000
```
4.3 Precedence of Properties

Since ORB properties can be obtained from various means, there is a precedence order in obtaining and using the properties. The following is a list of means to obtain properties. It is listed from highest to lowest precedence. If the same property is defined in two places, the place with the higher precedence will determine the property value.

- Command-line arguments
- Explicitly defined property file (defined by `-ORBconfig` argument or `com.twoab.orb2.config` property)
- Properties supplied programmatically
- Java System Properties
- Property file named `orb.properties` located in the JRE’s `lib` subdirectory

4.4 Command-Line Arguments

When a CORBA-based client or server application initializes an ORB, it passes the command-line arguments as a parameter to the initialization method. These command-line arguments are parsed during ORB initialization to determine if any are supported by the ORB’s implementation. Most command-line arguments have equivalent ORB property definitions. The following are command-line arguments that are supported by orb2 for Java.

The `-ORB` command-line arguments are consumed when passed to `ORB.init()`. That is, when the `ORB.init()` method returns, the arguments parameter will no longer contain the `-ORB` command-line arguments. If it is required to maintain these arguments (e.g. if you will call `ORB.init()` multiple time), you must make a copy of the argument array.

4.4.1 General Purpose

The following are general-purpose command-line arguments that can be used with client applications and/or services.

- `ORBbidir_iion <boolean>`
  Specifies a Boolean value that indicates if the bidirectional IIOP protocol should be used for all client-server connections. This should only be used in specialized environments that must traverse security firewalls. The default value is `false`.

- `ORBclient_request_interceptor_order <name_list>`
  Specifies a list of names of client interceptors (as returned by the interceptor's `name` method) in the order that the interceptors should be invoked. Commas should separate the names of the interceptors.

  **Note** It is not a requirement that the order of invocation for interceptors be specified.

- `ORBconfig <path>`
  Specifies the path of a property file that contains configuration properties used to initialize an ORB.

- `ORBdebug <boolean>`
  This command-line argument is used to indicate if the ORB should produce GIOP debug messages. The default value is `false`.

- `ORBdebug_file <path>`
  Specifies the path of a file where debug information is to be written. If debugging is activated and this argument is not specified, debug information will be directed to the standard output device (e.g. console).
4.4.1 General Purpose

Initializing the ORB

- **ORBgiop_minor_version**
  Specifies the minor version of the GIOP/IIOP protocol that objects in a server will use.

- **ORBhost <host_name>**
  Specifies the host name that will be used when creating interoperable object references for objects created using this ORB. The host name may be either a string-formatted host name or an internet address.

- **ORBid <orb_id>**
  Specifies a string that will be used to identify this ORB.

- **ORBInitRef <reference_info>**
  Specifies information about services whose object references can be obtained via the `resolve_initial_references` method. The reference information is in the form of:
  ```
  name=url
  ```
  where name is the name of the service that will be specified in `resolve_initial_references` (e.g. NamingService) and url is the URL formatted object reference. The orb2 product supports the "corbaloc," "ior" and "file" URL formats as specified by the OMG's CORBA specification. This command-line argument may be used multiple times when starting an application.

- **ORBmonitor <boolean>**
  Specifies a boolean value that determines whether or not this server should capture information required for monitoring. The default value is `false`.

- **ORBlicense_dir <directory containing license file>**
  Specifies the directory where the 2AB license file is located.

- **ORBmaxthreads <number_threads>**
  Specifies the maximum number of threads that may be started to handle CORBA requests. The default value, if not specified, is 20.

- **ORBminthreads <number_threads>**
  Specifies the minimum number of threads that may be started to handle CORBA requests. The default value, if not specified, is 1.

- **ORBport <port_number>**
  Specifies the port number that will be used to listens for incoming requests for objects created using this ORB.

- **ORBrequest_timeout <time_in_millis>**
  Specifies the time, in milliseconds, that a client will wait for a response from a request invoked on a service. If the timeout time specified is exceeded, the client that invoked the application will receive a TRANSIENT System Exception. The default value is 30,000 milliseconds.

- **ORBserver_request_interceptor_order <name_list>**
  Specifies a list of names of server interceptors (as returned by the interceptor's name method) in the order that the interceptors should be invoked. Commas should separate the names of the interceptors.

**Note**

It is not a requirement that the order of invocation for interceptors be specified.
4.4.2 Security

The following command-line arguments can be used when using the orb2 Security software. Please see the separate manual: Security Services User Guide for a complete description of all of orb2’s security features and how to use them.

-ORBthreadtimeout <timein_millis>
  Specifies the time, in milliseconds, that a thread will remain active without any activity. Specifically for services, a thread and any network socket connections it manages, will be terminated after the specified period of inactivity. The default value is 60,000 milliseconds.

-ORBsec <protocols>
  Specifies the security protocols to support. If the value is TLS, then only the transport layer will be used. If the value is TLS_UP, then both the transport layer and the GSSUP-based client authentication layer will be used. The TLS_CLIENT_ONLY value should be used for applications that are pure clients that can use only the transport layer. The TLS_UP_CLIENT_ONLY value should be used for applications that are pure clients that can use the transport and/or client authentication layers. If the value is UP, then the transport layer will not be used, only the GSSUP-based client authentication layer. The UP_CLIENT_ONLY value should be used for applications that are pure clients.

-ORBsec_debug <boolean>
  Specifies that the security software should produce verbose debug messages if the value is true.

-ORBsec_key_path
  Specifies the name or path of the keystore holding the client or service’s digital certificate. For the default Java 2 implementation, this is the path of the keystore file holding the certificate.

-ORBsec_key_pw <password_value>
  Specifies the password of the keystore holding the client or service’s digital certificate.

-ORBsec_port <port_number>
  Specifies the port that a secure service will listen for secure invocations.

-ORBsec_trust_path <path>
  Specifies the name or path of the keystore holding the certificates that are used to validate user or service certificates.

-ORBsec_trust_pw <password>
  Specifies the password of the keystore holding the certificates that are used to validate user or service certificates.

4.4.3 Trader Service

The following command-line arguments can be used when using the Trader Service.

-ORBprimary_trader <URL>
  Specifies URL for the primary orb2 Trader service to be used. The URL may be an "IOR" or "corbaloc" format. Specifies the Trader Service that will be returned from the resolve_initial_references method for the orb2_Trader.
4.5 ORB Properties

When an ORB is initialized, it uses configuration information from properties that are passed directly to the ORB.init method or properties that are obtained from special configuration files. Most property definitions have equivalent command-line arguments. The following sections describe the configuration properties that are supported by the orb2 for Java product.

4.5.1 General Purpose Properties

The following are general-purpose properties that can be used with client applications and/or services.

`com.twoab.orb2.bidir.iiop = <boolean>`

Specifies a Boolean value that indicates if the bidirectional IIOP protocol should be used for all client-server connections. This should only be used in specialized environments that must traverse security firewalls. The default value is `false`.

`com.twoab.orb2.char_set = <character set to use>`

Specifies the character set to use when using the `char` and `string` IDL data types. Valid values for this property are `default`, `ISO 8859-1` and `ISO-646`.

**Note** This property should only be used when using the RMI-IIOP protocol to interoperate with the ORB provided with JDK1.3. The only valid value in this case is `ISO-646`. Otherwise, the default value is `ISO 8859-1`. 
com.twoab.orb2.client.request.interceptor.order = <name_list>

Specifies a list of names of client interceptors (as returned by the interceptor's name method) in the order that the interceptors should be invoked. Commas should separate the names of the interceptors.

**Note** Specifying the order of invocation for interceptors is not required.

com.twoab.orb2.config = <path>

Specifies the path of a property file that contains configuration properties used to initialize an ORB.

com.twoab.orb2.debug.on = <boolean>

This property is used to indicate if the ORB should produce GIOP debug messages. The default value is false.

com.twoab.orb2.debug.file = <path>

Specifies the path of a file where debug information is to be written. If debugging is activated and this argument is not specified, debug information will be directed to the standard output device (e.g. console).

com.twoab.orb2.giop.minor.version

Specifies the minor version of the GIOP/IIOP protocol that objects in a server will use. The default value is 2.

com.twoab.orb2.host = <host_name>

Specifies the host name that will be used when creating interoperable object references for objects created using this ORB. The host name may be either a string-formatted host name or an internet address.

com.twoab.orb2.java.orb.properties = <boolean>

This is a boolean value indicating whether or not the orb.properties file located in the JRE’s lib subdirectory should be used as a source of properties to initialize an ORB. The default value of "true" indicates that the orb.properties file will be used as a source for properties. This property may only be passed as a Java System property or in the Properties object passed as an argument to ORB.init().

com.twoab.orb2.java.system.properties = <boolean>

This is a boolean value indicating whether or not the Java System Properties object should be used as a source of properties to initialize an ORB. The default value of "true" indicates that the Java System Properties object will be used as a source for properties. This property may only be passed as a Java System property or in the Properties object passed as an argument to ORB.init().

com.twoab.orb2.jthreads.max = <number_threads>

Specifies the maximum number of threads that may be started to handle CORBA requests. The default value, if not specified, is 20.

com.twoab.orb2.jthreads.min = <number_threads>

Specifies the minimum number of threads that may be started to handle CORBA requests. The default value, if not specified, is 1.

com.twoab.orb2.jthreads.timeout = <timein_millis>

Specifies the time, in milliseconds, that a thread will remain active without any activity. Specifically for services, a thread and any network socket connections it manages will be terminated after the specified period of inactivity. The default value is 60,000 milliseconds.
4.5.2 Security

The following properties can be used when using the orb2 Security software.

com.twoab.orb2.security

Specifies the security protocols to support. If the value is TLS, then only the transport layer will be used. If the value is TLS_UP, then both the transport layer and the GSSUP-based client authentication layer will be used. The TLS_CLIENT_ONLY value should be used for applications that are pure clients that can use only the transport layer. The TLS_UP_CLIENT_ONLY value should be used for applications that are pure clients that can use the transport and/or client authentication layers. If the value is UP, then the transport layer will not be used, only the GSSUP-based client authentication layer. The UP_CLIENT_ONLY value should be used for applications that are pure clients that use
only the client authentication layer. This is a required property for all secure clients and servers. This property may also be set using the –ORBsec command-line argument.

**com.twoab.orb2.security.access.control.operations**
This property is used to indicate whether access control for a service’s CORBA operations will be performed. The default value is true. This property can only be used with security option of TLS_UP or UP.

**com.twoab.orb2.security.audit.class**
Specifies the name of a Java class that should be used to create the implementation of the AuditChannel interface. This property is used to replace the default implementation that is part of the orb2 product.

**com.twoab.orb2.security.audit.file**
Specifies the path of the file that the default AuditChannel implementation will write to. If this property is not present, the default AuditChannel implementation will not write audit records.

**com.twoab.orb2.security.audit.invocation**
This property controls auditing operation invocations. A value of true indicates that auditing should be provided. A value of false indicates that auditing should not be done. The default value is true.

**com.twoab.orb2.security.audit.principal.auth**
This property controls auditing the authentication of principals. A value of true indicates that auditing should be provided. A value of false indicates that auditing should not be done. The default value is true.

**com.twoab.orb2.security.audit.session.auth**
This property controls auditing the server’s authentication of invoking principals. A value of true indicates that auditing should be provided. A value of false indicates that auditing should not be done. The default value is true.

**com.twoab.orb2.security.audit.success**
Specifies the rule to use for recording successful security events (e.g. successful authentication). A value of true indicates that successful events should be recorded. A value of false indicates that successful events should not be recorded. The default value is true.

**com.twoab.orb2.security.authenticate.client**
Specifies a value (true or false) indicating whether or not the service requires clients to provide proof of authentication. If this property is not specified, a default value of false will be used.

**com.twoab.orb2.security.authentication.class**
This specifies a user-defined Java class that is used to satisfy authentication requests made by the CSIv2 GSSUP feature and/or the security service PrincipalAuthenticator interface. This class must implement the com.twoab.orb2.pub.security.Authentication interface.

**com.twoab.orb2.security.center.host**
The host name where the iLock Security Center is located. This is used for the integration of GSSUP authentication with iLock.

**com.twoab.orb2.security.center.port**
The port number used by the iLock Security Center. This is used for the integration of GSSUP authentication with iLock.
4.5.2 Security

Initializing the ORB

.com.twoab.orb2.security.cipher.suites
Specifies one or more cipher suites to be the enabled cipher suites for use in creating TLS servers. The cipher suites must be separated by a ‘,’ character. The enabled cipher suites must be among the supported cipher suites for vendors TLS implementation.

.com.twoab.orb2.security.create.identity
Specifies a value (true or false) indicating whether or not an intermediate service will create identity tokens when the intermediate server has authenticated the identity. If this property is not specified, a default value of true will be used.

.com.twoab.orb2.security.debug
If this property is set to a value of true, verbose messages will be produced that describe the CSIv2 protocol activity that takes place. This property may also be set using the –ORBsec_debug command-line argument.

.com.twoab.orb2.security.instance
Applications (both clients and services) that implement the Client Authentication Layer must have access to an instance of the iLock Security Center. An instance name is used to identify an instance of an iLock Security Center. If this property is not set, a default value of default will be used.

.com.twoab.orb2.security.key.algorithm
Defines the key manager algorithm to use. The default value is "SunX509".

.com.twoab.orb2.security.key.provider.class
Specifies the provider class for key management. The default value is com.sun.net.ssl.KeyManagerFactory.

.com.twoab.orb2.security.key.store.password
Specifies the password of the keystore holding the client or service’s digital certificate. This is a required property for all services that use the TLS/SSL protocol. Client programs only supply this property if they are required to do client authentication. This property may also be set using the –ORBsec_key_pw command-line argument.

.com.twoab.orb2.security.key.store.path
Specifies the name or path of the keystore holding the client or service’s digital certificate. For the default Java 2 implementation, this is the path of the keystore file holding the certificate. This is a required property for all services that use the TLS/SSL protocol. Client programs only supply this property if they are required to do client authentication. This property may also be set using the –ORBsec_key_path command-line argument.

.com.twoab.orb2.security.key.store.provider.class
Specifies the provider class for key store management. The default value is java.security.KeyStore.

.com.twoab.orb2.security.key.store.type
Specifies the key store type. The default value is JKS.

.com.twoab.orb2.security.name
This property may be specified to create a globally unique name for a service that will support CSIv2. If the property is not set, orb2 for Java will generate a globally unique name.
com.twoab.orb2.security.password

Client application must specify the User ID and Password to be used to authenticate its user. This property may be used to specify the user’s password. Typically, this property should not be placed in a configuration file, but rather provided programmatically by the client application. An alternative, and perhaps a more standard technique, would be to use the CORBA Sec PrincipalAuthenticator interface (see section that describes the PrincipalAuthenticator interface).

If this property is set for an intermediate server, then this password and the ID specified by com.twoab.orb2.security.user will be passed to the target service rather than an identity token.

com.twoab.orb2.security.port

Specifies the port that a secure service will listen for secure invocations. If this property is not specified, a default port number of 684 will be used. This property may also be set using the –ORBsec_port command-line argument.

com.twoab.orb2.security.stateless

By default, the Client Authentication Layer will use statefull connections. The term statefull indicates that client authentication and authorization is only done for the first message sent after establishing a session between a client and a service. Since the authentication/authorization process involves significant overhead, this statefull behavior is obviously desirable. There are some cases (e.g. stateless Enterprise Java Beans) where there is no concept of a session or state. The Client Authentication Layer software is capable of negotiating state and hence for CORBA applications there is typically no reason to force stateless behavior; however, this property can be set to a value of true to force a client or service to use stateless behavior.

com.twoab.orb2.security.trust.algorithm

Defines the trust manager algorithm to use. The default value is "SunX509" for most platforms. For IBM AIX it is IBMX509.

com.twoab.orb2.security.trust.identity

Specifies a value (true or false) indicating whether or not the service will trust identity tokens created by intermediate target servers. The intermediate server has authenticated the identity. If this property is not specified, a default value of true will be used.

com.twoab.orb2.security.trust.provider.class

Specifies the provider class for trust management. The default value is com.sun.net.ssl.KeyManagerFactory.

com.twoab.orb2.security.trust.store.password

Specifies the password of the “keystore” holding the certificates that are used to validate user or service certificates. This is a required property for all clients that use the TLS/SSL protocol. Service programs only supply this property if they require client authentication. This property may also be set using the –ORBsec_trust_pw command-line argument.

com.twoab.orb2.security.trust.store.path

This property specifies the name or path of the keystore holding the certificates that are used to validate user or service certificates. This is a required property for all clients that use the TLS/SSL protocol. Service programs only supply this property if they require client authentication. This property may also be set using the –ORBsec_trust_path command-line argument.

com.twoab.orb2.security.trust.store.provider.class

This specifies the provider class for trust store management. The default value is java.security.KeyStore.
4.5.3 Trader Service

The following properties can be used when using the Trader Service.

**com.twoab.orb2.trader.hostname = <host_name>**

Specifies the host name or internet address of the Trader Service that will be returned from the `resolve_initial_references` method for the orb2_Trader. The deprecated property, `com.twoab.orb2.Trader.1.hostName`, may also be used.

**com.twoab.orb2.trader.port = <port_number>**

Specifies the port number of the Trader Service that will be returned from the `resolve_initial_references` method for the orb2_Trader. The default value is 11002. The deprecated property, `com.twoab.orb2.Trader.1.port`, may also be used.

**com.twoab.orb2.trader2.hostname = <host_name>**

Specifies the host name or internet address of the Trader Service that will be returned from the `resolve_initial_references` method for the orb2_SecondaryTrader. The deprecated property, `com.twoab.orb2.Trader.2.hostName`, may also be used.

**com.twoab.orb2.trader2.port = <port_number>**

Specifies the port number of the Trader Service that will be returned from the `resolve_initial_references` method for the orb2_SecondaryTrader. The default value is 11001. The deprecated property, `com.twoab.orb2.Trader.2.port`, may also be used.
Chapter 5  orb2 Security Features

5.1 Common Secure Interoperability Version2 (CSIv2)

The orb2 for Java product provides an implementation of CSIv2 that conforms to “Conformance Level 0.” It supports the TLS/SSL Transport Layer and the SAS - Client Authentication Layer and Identity Assertion in the Security Attribute Layer. It does this for both stateless and statefull connections. The SAS - Client Authentication Layer provides the additional support for access control for CORBA operation invocations. Whenever one of the CSIv2 layers detect a security violation, a CORBA NO_PERMISSION exception will be thrown with a minor code value that indicates the reason.

In order to meet a wide variety of security requirements, the security functionality provided by orb2 for Java is highly configurable. Security functionality is configured via properties that are processed during ORB initialization. These properties may be specified in a property file or passed to the ORB::init method. See the orb2 for Java User Manual - Initialising the orb, for details regarding the use of ORB properties and the specific properties used to configure the CSIv2 implementation.

The orb2 Security Services User Guide describes the CSIv2 implementation in detail.

5.2 CORBA Security Services (CSS)

The orb2 for Java CORBA Security Service (CSS) enables CORBA-based services to be secured. This component provides security service software that runs in the same process with CORBA-based clients and/or servers. It provides the basic services that allow servers to authenticate their clients and control access to CORBA operations and objects.

The CSS component uses the orb2 Security Center to manage information about users, security policies, interfaces and objects and the association of security policies with interface operations and object domains. There can be multiple instances of the Security Center that each control a specified domain of users, policies, and so forth.

The orb2 Security Services User Guide describes the CSS implementation in detail.

5.3 Resource Access Decision Facility

orb2 for Java provides a full implementation of the OMG’s Resource Access Decision Facility Security Specification. This facility provides a framework in support of fine-grain access controls required by "application-level" security. In addition to necessary run-time components, the RAD also provides application programming interfaces that ease the burden of introducing security into business applications.

5.4 Auditing

Neither CSIv2 nor CORBA Security Services provide a standard for security auditing. By default, the orb2 for Java's implementation of CSIv2 and CSS provide a simple proprietary sequential audit file recording format. If the audit file does not exist, a new file will be created. If the file exists, records will be appended to the existing file.

The auditing feature supports three auditing points. They are the principal authentication point, the service authorization point (where a service authenticates a client) and the operation access control point. The auditing points can be controlled via ORB properties. The control of successful and failed security events can be configured. The orb2 Security Services User Guide describes the auditing implementation in detail.

5.5 Programming API’s

The standard programming APIs provided are defined by the OMG security standards: Common Secure Interoperability Version 2 (CSI/v2), CORBA Security Service (CSS), and the Resource Access Decision Facility (RAD) specifications. The standard CORBA interfaces are documented in the OMG specifications and defined in OMG IDL.

The CORBA Security Service API’s are specified in the orb2 Security Services User Guide. The Resource Access Decision Facility IDL-defined CORBA API’s are fully supported. In addition, a set of CORBA Convenience Classes for the Java programmer has also been defined. These classes simplify the development of RAD client applications. Reference documentation for the interfaces and convenience classes is provided in the Resource Access Decision Facility User Manual.

5.6 The Security Center

The Security Center manages information about users, secured resources, and access policy. The Security Center service may reside on any machine within an enterprise. Applications do not directly use the Security Center service but rather use CSIv2 and/or the CORBASEC APIs which interface with the Security Center. Multiple applications, running on different machines, may concurrently utilize the Security Center.


5.7 Identity and Policy Administration Tools

orb2 provides a choice of administrative tools.

The orb2 Security Services User Guide describes the Administrative tools in detail.

5.8 orb2 Security Manuals

5.8.1 orb2 Security Services User Guide

The orb2 Security Services User Guide is located in the docs sub-directory of your orb2 for Java installation.

5.8.2 Resource Access Decision Facility User Guide

The Resource Access Decision Facility User Guide is located in the docs sub-directory of your orb2 for Java installation.
An Object Adapter specifies how server CORBA objects interact with the ORB. The Portable Object Adapter (POA) interface was defined to improve portability and provide increased functionality to server applications.

The POA performs the following functions:

- Demultiplexes each client request to the appropriate servant.
- Dispatches the requested operation to the target servant, using the skeleton to transform the parameters in the request into arguments that are passed to the intended servant operation.
- Activates and deactivates objects and incarnates and etherealizes their corresponding servants, as necessary. Servants are objects that implement the requests on one or more objects.
- Generates object references for the objects that are registered with it. The object references contain addressing information on how to reach that object in the distributed system.

Figure 6.1 shows how the POA interacts with a servant and the ORB.
The POA supports both static and dynamic (DSI) skeletons. The POA interface provides operations to manage objects and POAs as summarized in Table 6.1.

Table 6.1 POA Interface Operations

<table>
<thead>
<tr>
<th>Functional Category</th>
<th>Operations Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>POA Lifecycle Management</td>
<td>create_POA, find_POA, destroy</td>
</tr>
<tr>
<td>Policy Creation</td>
<td>create_policy_name</td>
</tr>
<tr>
<td>Attributes</td>
<td>the_name, the_parent, the_POAManager, the_activator</td>
</tr>
<tr>
<td>Servant Manager Operations</td>
<td>get_servant_manager, set_servant_manager</td>
</tr>
<tr>
<td>Default Servant Operations</td>
<td>get_servant, set_servant</td>
</tr>
<tr>
<td>Object Activation/Deactivation</td>
<td>activate_object, activate_object_with_id, deactivate_object</td>
</tr>
<tr>
<td>Object Reference Creation</td>
<td>create_reference, create_reference_with_id</td>
</tr>
<tr>
<td>Conversions</td>
<td>servant_to_id, servant_to_reference, reference_to_servant, reference_to_id, id_to_servant, id_to_reference</td>
</tr>
</tbody>
</table>

All servers contain at least one distinguished POA called the Root POA that the ORB manages. An application can also create and activate additional POAs, which are nested as child POAs from the Root POA (see Figure 6.2). Each POA provides a namespace for object IDs and a namespace for other nested POAs. A single server process may contain a hierarchy of nested POA instances, forming a hierarchical name space for objects within a server. If the POA supports a policy for retaining servants, it sets up an Active Object Map, which is a list that maps active object references to associated servants.

To use the POA, an application performs the following steps:

1. Gets or creates a new POA.
2. Acquires a servant object.
3. Activates the object.
4. Activates the POA manager.
5. Deactivates the object.

The following sections describe each step.

6.1 Creating a POA

You must specify which object adapter you want to use for interfacing objects with the ORB. The POA interface provides a Root POA and also allows custom POAs. The Root POA is provided by the ORB and is available to every application. Applications can also create or use other POAs. Both approaches are addressed in the following sections.
6.1.1 Using the Root POA

The Root POA is a distinguished POA object that is managed by the ORB. It is available to every application via the ORB `resolve_initial_references()` operation, using the object name `RootPOA`. All server applications must obtain a reference to the `RootPOA`, either to use it directly to manage objects or to create new POA objects.

Example

Application server obtains a reference to the `RootPOA`. The `Helper` class narrows the `org.omg.CORBA.Object` type returned from `resolve_initial_references` to an `org.omg.PortableServer.POA` type

```java
// first, initialize the ORB
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
// then obtain an object reference for the RootPOA
org.omg.PortableServer.POA rootPOA =
    org.omg.PortableServer.POAHelper.narrow (orb.resolve_initial_references("RootPOA"));
```
Associated with a POA is a number of policies that determine the characteristics of the objects implemented in that POA. The policies associated with a POA are specified when the POA is created. You cannot change them after it is created.

**Table 6.2 Default Policies for the Root POA**

<table>
<thead>
<tr>
<th>Policy</th>
<th>Value Attribute for RootPOA</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>ORB_CTRL_MODEL</td>
<td>ORB is responsible for assigning requests to threads.</td>
</tr>
<tr>
<td>Lifespan</td>
<td>TRANSIENT</td>
<td>Objects implemented in the POA cannot outlive the process in which they are created. After the POA is deactivated, the use of any object references generated from it results in an OBJECT_NOT_EXIST exception.</td>
</tr>
<tr>
<td>ObjectId Uniqueness</td>
<td>UNIQUE_ID</td>
<td>Servants activated with the POA support one object ID.</td>
</tr>
<tr>
<td>ID Assignment</td>
<td>SYSTEM_ID</td>
<td>Objects created with the POA are assigned object IDs only by the POA.</td>
</tr>
<tr>
<td>Servant Retention</td>
<td>RETAIN</td>
<td>The POA retains active servants in its Active Object Map.</td>
</tr>
<tr>
<td>Request Processing</td>
<td>USE_ACTIVE_OBJECT_MAP_ONLY</td>
<td>If the object ID is not found in the Active Object Map, an OBJECT_NOT_EXIST exception is returned to the client.</td>
</tr>
<tr>
<td>Implicit Activation</td>
<td>IMPLICIT_ACTIVATION</td>
<td>The RootPOA supports implicit activation of servants.</td>
</tr>
</tbody>
</table>

The POA policies also have interdependence with parameters that can be passed into org.omg.CORBA.ORB.init when the ORB is initialized. For example, you can control the threadpool that the RootPOA uses with the ORB_CTRL_MODEL thread policy. Chapter 4, Initializing the ORB, describes the parameters you can use with the init() method and each POA policy.

### 6.1.2 Using a Custom POA

Creating a new POA allows you to control the creation and behavior of objects created with that POA. Create a new POA if you need to accomplish one of the following:

- Declare policy choices different from those for the RootPOA.
- Use an adapter activator or servant manager.
- Partition the namespace of objects — object IDs are interpreted relative to a POA.
- Independently control request processing for multiple sets of objects.

A POA is created as a child of an existing POA using the POA::create_POA operation on the parent POA, which forms a hierarchy starting with the Root POA (see Figure 6.2).

The create_POA() operation has the following signature

```
POA create_POA(
    in string adapter_name,
    in POAManager a_POAManager,
    in CORBA::PolicyList policies)
raises (AdapterAlreadyExists, InvalidPolicy);
```

This operation creates a new POA as a child of the target POA. The specified name identifies the new POA with respect to other POAs with the same parent POA; the name must be unique to all other POAs with the same parent. The AdapterAlreadyExists exception is raised if the target POA already has a child POA with the specified name.
If the `a_POAManager` parameter is null, a new `POAManager` object is created and associated with the new POA. Otherwise, the specified `POAManager` object is associated with the new POA. The `POAManager` object is obtained using the attribute name `the_POAManager`. See Section 6.4, Activating the POA Manager.

The `policies` parameter to the `create_POA` operation specifies the policies for that POA. The policy objects control the POA. You cannot change it after the POA is created. Policies are not inherited from the parent POA.

**Examples**

Create a POA named `my_little_poa` with a `ORB_CTRL_MODEL` thread policy and `TRANSIENT` Lifespan policy.

```java
// create the policies
org.omg.CORBA.Policy[] policies = new org.omg.CORBA.Policy[2];
policies[0] = rootPOA.create_thread_policy(
    org.omg.PortableServer.ThreadPolicyValue.ORB_CTRL_MODEL);
policies[1] = rootPOA.create_lifespan_policy(
    org.omg.PortableServer.LifespanPolicyValue.TRANSIENT);

// create a new POA
org.omg.PortableServer.POA poa = rootPOA.create_POA( "my_little_poa",
                                      null,
                                      policies );
```

Create a POA named `calcPOA` that has a Lifespan policy of `PERSISTENT` and an ID Assignment policy of `USER_ID`. This allows the Calculator application to assign IDs to objects created with this POA. The objects can outlive the process in which they are created.

```java
org.omg.CORBA.Policy[] policy_list =
    new org.omg.CORBA.Policy[2];
policy_list[0] = poa.create_id_assignment_policy(
    org.omg.PortableServer.IdAssignmentPolicyValue.USER_ID);
policy_list[1] = poa.create_lifespan_policy(
    org.omg.PortableServer.LifespanPolicyValue.PERSISTENT);

org.omg.PortableServer.POA calcPOA =
    poa.create_POA("calcPOA",
                   poa.the_POAManager(),
                   policy_list );
```

The POA policies, which are summarized in Table 6.3, allow you to control the behaviors.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Value Attributes</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td><code>ORB_CTRL_MODEL</code></td>
<td>(default) Each POA contains a thread pool, which is specified in the ORB.init parameters. The ORB is then responsible for assigning requests to threads.</td>
</tr>
<tr>
<td></td>
<td><code>SINGLE_THREAD_MODEL</code></td>
<td>Requests in the POA are processed sequentially. Beware of deadlock situations, such as expecting a callback on a single thread.</td>
</tr>
<tr>
<td>Lifespan</td>
<td><code>TRANSIENT</code></td>
<td>(default) Objects implemented in the POA cannot outlive the process in which they are created.</td>
</tr>
<tr>
<td></td>
<td><code>PERSISTENT</code></td>
<td>Objects implemented in the POA can outlive the process in which they are first created.</td>
</tr>
</tbody>
</table>
For most applications, the Request Processing policy is one of the most critical policy decisions. After determining the appropriate value to use for this policy, look at the other policies appropriate for the objects that POA contains. Combinations of the Servant Retention and Request Processing policies are powerful in controlling the behavior of objects created within the POA as described in Table 6.4.

**Table 6.3 POA Policies (Continued)**

<table>
<thead>
<tr>
<th>Policy</th>
<th>Value Attributes</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ObjectId</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Uniqueness</strong></td>
<td><strong>UNIQUE_ID</strong></td>
<td>(default) Servants activated with the POA support one object ID.</td>
</tr>
<tr>
<td><strong>MULTIPLE_ID</strong></td>
<td></td>
<td>Servants activated with the POA may support one or more object IDs.</td>
</tr>
<tr>
<td><strong>ID Assignment</strong></td>
<td><strong>SYSTEM_ID</strong></td>
<td>(default) Objects created with the POA are assigned object IDs only by the POA.</td>
</tr>
<tr>
<td><strong>USER_ID</strong></td>
<td></td>
<td>The application assigns object IDs for objects created with this POA. If the POA also has the PERSISTENT policy; you must use unique object IDs across all instantiations of the same POA.</td>
</tr>
<tr>
<td><strong>Servant Retention</strong></td>
<td><strong>RETAIN</strong></td>
<td>(default) The POA retains active servants in its Active Object Map.</td>
</tr>
<tr>
<td><strong>NON_RETAIN</strong></td>
<td></td>
<td>Servants are not retained in the POAs Active Object Map.</td>
</tr>
<tr>
<td><strong>Request Processing</strong></td>
<td><strong>USE_ACTIVE_OBJECT_MAP_ONLY</strong></td>
<td>(default) If the object ID is not found in the Active Object Map, an OBJECT_NOT_EXIST exception is returned to the client. Also requires the RETAIN policy.</td>
</tr>
<tr>
<td><strong>USE_DEFAULT_SERVANT</strong></td>
<td></td>
<td>If the object ID is not in the Active Object Map or the NON_RETAIN policy is in effect, the request is dispatched to the default servant registered with the POA using set_servant. Also requires the MULTIPLE_ID policy.</td>
</tr>
<tr>
<td><strong>USE_SERVANT_MANAGER</strong></td>
<td></td>
<td>If the object ID is not in the Active Object Map or the NON_RETAIN policy is in effect, the servant manager, registered with the POA using set_servant_manager, is given the opportunity to locate a servant.</td>
</tr>
<tr>
<td><strong>Implicit Activation</strong></td>
<td><strong>IMPLICIT_ACTIVATION</strong></td>
<td>POA supports implicit activation of servants. Also requires the RETAIN and SYSTEM_ID policies.</td>
</tr>
<tr>
<td><strong>NO_IMPLICIT_ACTIVATION</strong></td>
<td>(default)</td>
<td>POA does not support implicit activation of servants.</td>
</tr>
</tbody>
</table>

For most applications, the Request Processing policy is one of the most critical policy decisions. After determining the appropriate value to use for this policy, look at the other policies appropriate for the objects that POA contains. Combinations of the Servant Retention and Request Processing policies are powerful in controlling the behavior of objects created within the POA as described in Table 6.4.

**Note**  The POA policies also have interdependencies with parameters that are passed into org.omg.CORBA.ORB.init when the ORB is initialized. For example, you can set ORB-wide parameters that control the threadpool for POAs created with the ORB_CTRL_MODEL thread policy and ORB-wide parameters that affect POAs that use the PERSISTENT and USER_ID policies. Refer to Chapter 4, Initializing the ORB, for details about the parameters you can use with init().

If you create a POA with an incompatible set of policies, an exception is raised when the POA tries to create an object. The policies that are supported for the POA can affect how operations behave; you must ensure that the POA policies are valid for the operation you are calling. Refer to Chapter 11, Miscellaneous Tools, for more information about the POA policies.

### 6.2 Servant Objects

The servant object is code that implements requests on one or more objects. Servants usually exist within the context of a server process. On creation, the servant must be registered with a POA before a remote client can access it.
Creating a servant is called *incarnation*; destroying a servant is known as *etherealization*. Incarnation associates a servant with an object so that it may service requests. Etherealization destroys that association.

The POA supports dynamic servants and two types of static servants

- **Inheritance-based Servants**
  Drive their class from an abstract skeleton base class. The POA upcalls these servants via the servant inheritance hierarchy.

- **Delegation (Tie-based) Servants**
  Tie-based servants are connected to the POA by a servant *Tie* object that is automatically generated by the orb2 IDL compiler. The POA upcalls these servants by the *Tie* object delegating to the servant.

The `demos/calc` subdirectory that contains sample programs for the Calculator application includes both an inheritance-based servant (`CalcServer.java`) and a Tie-based servant (`CalcServer_t.java`).

For some applications, for example, those that contain thousands of objects, explicitly registering a servant for each object is not realistic or scalable. Applications can use servant managers to activate objects on demand as requests are made on them instead of activating them all at startup. Another alternative is to have a single default servant that uses the `USE_DEFAULT_SERVANT` policy and implements all the objects on a POA.

The `PortableServer::Current` interface, which is derived from `CORBA::Current`, is available to support servants that implement multiple objects. You can also use it within the context of POA-dispatched method invocations on any servant. The `Current` interface provides method implementations with access to the identity of the object on which the method was invoked.

**Example**

```java
interface Current : CORBA::Current {
  exception NoContext();
  POA get_POA();
    raises (NoContext);
  ObjectId get_object_id();
    raises (No Context);
};
```

The `get_POA()` operation returns a reference to the POA that is implementing the object in whose context it is called. If you call this operation outside the context of a POA-dispatched operation, a `NoContext` exception is raised.

The `get_object_id()` operation returns the `ObjectId` that identifies the object in whose context it is called. A `NoContext` exception is raised if this operation is called outside the context of a POA-dispatched operation.

---

**Table 6.4 Servant Retention and Request Processing Policy Behaviors**

<table>
<thead>
<tr>
<th>Request Processing</th>
<th>Servant Retention</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>USE_ACTIVE_OBJECT_MAP</strong></td>
<td>Must explicitly activate all objects.</td>
</tr>
<tr>
<td><strong>USE_DEFAULT_SERVANT</strong></td>
<td>POA invokes a default servant if it does not find a servant in the Active Object Map.</td>
</tr>
<tr>
<td><strong>USE_SERVANT_MANAGER</strong></td>
<td>POA invokes the specified ServantManager if it does not find a servant in the Active Object Map.</td>
</tr>
</tbody>
</table>
6.2.1 **Servant Managers**

Servant managers are responsible for managing the association of an object (the *ObjectId* value) with a particular servant and determining if an object exists. A POA configured with the `USE_SERVANT_MANAGER` policy relies on an application-supplied `ServantManager` object to supply the object/servant associations. A servant manager implementation must provide

- Implementation code for either
  - Activating the servant using `incarnate()`
  - Locating the servant using `preinvoke()`
- Implementation code for the servant operations (as for all servants)

Servant managers have two standard interfaces that are derived from the `ServantManager` interface. If the POA has the `RETAIN` value for the Servant Retention policy, the servant manager needs to supply the `ServantActivator` interface. If the POA does not maintain an Active Object Map because it uses the `NON_RETAIN` policy value, the servant manager must supply the `ServantLocator` interface.

A `ServantManager` object reference is implemented as a Java reference to the `ServantManager` implementation. To acquire a `ServantManager` object reference from the `ServantManager` implementation, call `_this()` or `_this(ORB)`.

### 6.2.1.1 ServantActivator Interface

When the POA needs a servant to incarnate the target object, it invokes the `incarnate` operation. Also, when it needs to deactivate an object or destroy a servant, it invokes the `etherealize` operation. Servants are etherealized when the POA is destroyed or when a `deactivate_object` or ORB `shutdown` operation is called.

**Example**

```java
interface ServantActivator: ServantManager {
    Servant incarnate(
        in ObjectId oid,
        in POA adapter)
        raises (ForwardRequest);

    void etherealize(
        in ObjectId oid,
        in POA adapter,
        in Servant the_servant,
        in boolean cleanup_in_progress,
        in boolean activations);
}
```

A `ServantActivator` object reference is implemented as a Java reference to the `ServantActivator` implementation. To obtain a `ServantActivator` object reference for a particular `ServantActivator` implementation, call `_this()` or `_this(ORB)`.

**Example**

```java
ServantActivator that activates objects on demand. It assumes that a POA has the USER_ID, USE_SERVANT_MANGER and RETAIN policies.

class MyFooServantActivator extends org.omg.PortableServer.ServantActivatorPOA {
    public org.omg.PortableServer.Servant incarnate (byte[] oid, 
        org.omg.PortableServer.POA poa ) throws 
        org.omg.PortableServer.ForwardRequest 
    { 
```
//if the object id equals "myLittleFoo" then create a
//new MyFooServant servant object
String s = new String( oid );
if ( s.compareTo( new String( "myLittleFoo" )) == 0 )
   return new MyFooServant( poa, 42 );
else
   throw new org.omg.CORBA.OBJECT_NOT_EXIST();
}
public synchronized void etherealize ( byte[] oid,
   org.omg.PortableServer.POA adapter,
   org.omg.PortableServer.Servant serv,
   boolean cleanup_in_progress,
   boolean remaining_activations )
{
   if ( remaining_activations == false )
      serv = null;
}

6.2.1.2 ServantLocator Interface

When a POA uses the NON_RETAIN value for the ServantRetention policy, it does not maintain an Active Object Map of the object/servant associations. Therefore, the POA expects a servant manager that inherits from the ServantLocator interface instead of from ServantActivator.

A ServantLocator object reference is implemented as a Java reference to the ServantLocator implementation. To acquire a ServantLocator object reference for a particular ServantLocator implementation, call _this() or _this(ORB).

Like a ServantActivator, the ServantLocator returns a servant and the POA dispatches the request to it. However, the semantics for ServantLocator differ because they are invoked for every request on an object. The POA invokes preinvoke before dispatching a request to obtain a servant and invokes postinvoke after the servant handles the request. The preinvoke and postinvoke methods are inherited from the PortableServer::ServantLocator POA skeleton class.

Example

interface ServantLocator:ServantManager{
   native Cookie;
   Servant preinvoke(
      in ObjectId id,
      in POA adapter)
      in CORBA::Identifier operation,
      out Cookie the_cookie);
   raises (ForwardRequest);
   void postinvoke)
   in ObjectId id
   in POA adapter
   in CORBA::Identifier operation,
   in Cookie the_cookie,
   in Servant the_servant);
};

Although they perform similar functions, the signatures for the preinvoke and postinvoke methods differ from the incarnate and etherealize methods. Since the servant returned by a ServantLocator is used to process only a single request at a time, it can pass additional parameters.
With the **preinvoke** method, you can pass the name of the operation that is being invoked on the target object, which allows the **ServantLocator** to return a different servant, depending on which operation is being invoked. You also can pass a cookie, which the POA subsequently passes back to the **postinvoke** method. This allows the **ServantLocator** to attach state to each pair of **preinvoke** and **postinvoke** invocations. The definition of the cookie is defined by the application and the POA passes it unchanged.

The **postinvoke** method must complete before the request on an object is considered complete. If the method finishes normally but **postinvoke** raises a system exception, the method’s normal return is overridden with the exception.

### 6.2.2 Default Servants

Default servants allow applications to supply servants for use with the Dynamic Skeleton Interface (DSI), which provides a single **invoke** function that can dispatch any request. You also can use a default servant based on static skeletons if all the objects incarnated by the default servant support the same interface. By using the **USE_DEFAULT_SERVANT** policy, you can create a POA that uses a single servant to implement all its objects.

To use a default servant, create a POA with **USE_DEFAULT_SERVANT** for the Servant Retention policy. If no other servant is available —because the servant is not found in the Active Object Map or the POA has the **NON_RETAIN** policy value— the POA invokes the specified default servant regardless of the object ID specified for the target object.

You can set and obtain the default servant using the operations defined for the **USE_DEFAULT_SERVANT** policy.

**Example**

```java
Servant get_servant()
    raises (NoServant, WrongPolicy);

void set_servant(in Servant p_servant)
    raises (WrongPolicy);
```

The **get_servant** or **set_servant** operations must complete successfully to set or obtain the servant.

### 6.2.3 Adapter Activators

An adapter activator is a user-defined callback object that allows the POA to activate a child POA on demand. This can occur when a request is received that names an inactive child POA, or when **find_POA** is called on an inactive child POA and the **activate** parameter value is "true." An adapter activator is necessary only when creating POAs during request processing, not when an application server creates all its needed POAs at the beginning of execution.

**Example**

```java
interface AdapterActivator {
    boolean unknown_adapter(
        in POA parent,
        in string name);
};
```

While a request from the POA to an adapter activator is in progress, all requests to objects managed by the new POA, or any descendant POAs, are queued. This serialization allows the adapter activator to complete initialization of the new POA before requests are delivered to it. An **AdapterActivator** object must be local to the process containing the POA objects to which it is registered.

An **AdapterActivator** object reference is implemented as a Java reference to the **AdapterActivator** implementation. To obtain an **AdapterActivator** object reference for a particular **AdapterActivator** implementation, call **_this()** or **_this(ORB)**.
Example

Implement a POA Activator. The **unknown_adapter** function checks the name of the adapter to create. If the name is **my_little_poa**, it creates the POA and returns "true." The **unknown_adapter** function also attaches a **ServantActivator** to the created POA, which creates servants on demand.

class ChildPOAActivator extends org.omg.PortableServer.AdapterActivatorPOA {
    public boolean unknown_adapter(
        org.omg.PortableServer.POA parent,
        java.lang.String name )
    {
        try {
            if ( name.compareTo("my_little_poa") == 0 )
            {
                //create the policies
                org.omg.CORBA.Policy[] policies = new org.omg.CORBA.Policy[3];
                policies[0] = parent.create_id_assignment_policy(
                                IdAssignmentPolicyValue.USER_ID);
                policies[1] = parent.
                                create_request_processing_policy(
                                RequestProcessingPolicyValue.USE_SERVANT_MANAGER);
                policies[2] = parent.create_servant_retention_policy(
                                ServantRetentionPolicyValue.RETAIN);
                //create a new POA
                org.omg.PortableServer.POA poa = parent.create_POA("my_little_poa",
                                parent.the_POAManager(),
                                policies);
                //and create a ServantActivator and attach it to
                //created POA
                MyFooServantActivator fooIM = new MyFooServantActivator();
                org.omg.PortableServer.ServantActivator IMref = fooIM._this();
                poa.set_servant_manager(IMref);
                return true;
            }
            else {
                // request to create POA with another name
            }
        }
        catch ( Exception e )
        {
            System.out.println("Exception occurred = " +
                                e.toString());
        }
    }
System.out.println( "POA Activator returning false" );
return false;
}
}

This excerpt shows the initialization code required to assign the previous example POA activator with the RootPOA.
ChildPOAActivator poa_activatorIM =
    new ChildPOAActivator();
org.omg.PortableServer.AdapterActivator poa_activator =
    poa_activatorIM._this();
rootPOA.the_activator( poa_activator );

6.3 Activating Objects

CORBA objects and servants have different lifetimes. A CORBA object is created, then activated, either immediately or when a request is received. When an object is activated, it can be incarnated by a servant, or during its activation period, multiple servants may be incarnated and etherealized for each request. Therefore, a single CORBA object may be represented by one or more servants over its lifetime, and a servant may represent one or more CORBA objects simultaneously.

The lifetime of a CORBA object is controlled by the Lifespan policy of the POA that implements the object. If the POA has a Lifespan policy of TRANSIENT, the lifetime of the object cannot outlive the process that creates it. If the POA has a Lifespan policy of PERSISTENT, the lifetime of the object is independent of the lifetime of any server processes in which they are activated.

6.3.1 Identifying Objects

To process a request, it must be able to identify the target object as well as the POA that created the target object reference. An object identifier (ObjectId) identifies a CORBA object within the context of a POA.

Example

//object activation
ObjectId activate_object(
    in Servant p_servant)
    raises (ServantAlreadyActive, WrongPolicy);
void activate_object_with_id(
    in ObjectId id,
    in Servant P_servant)
    raises (ServantAlreadyActive, ObjectAlreadyActive, WrongPolicy);

The ObjectId provides the mechanism by which client requests are associated with CORBA objects. Applications or the POA can assign the ObjectId. Since the ObjectId must be unique only within the context of its POA, they are not guaranteed to be unique global identifiers.

CORBA objects can be transient or persistent, depending on the Lifespan policy of the POA in which they are implemented. Since transient objects cannot outlive the process that creates the object, the ObjectId for the object is valid only until the process exits. If a client attempts to invoke an operation using an ObjectId for a transient object whose process has exited, the CORBA::OBJECT_NOT_EXIST exception is raised. If a client needs to invoke operations using the same ObjectId, regardless of whether the server process exits and restarts, the object must be implemented using a POA that has the PERSISTENT Lifespan policy.
When a client issues a request, the ORB first locates an appropriate server (starting one if needed) and then locates the appropriate POA within that server. After it locates the appropriate POA, the ORB delivers the request to that POA. Processing of the request depends upon the policies associated with the POA as well as the object’s current state of activation.

The POA activation facilities involve activating the object and incarnating a servant. The POA supports the following modes of activation, depending on the attribute values specified for the POA’s Servant Retention and Activation policies. If the POA has the RETAIN policy, the servant and its associated ObjectId is entered into the Active Object Map of the appropriate POA. In this case, you can make the activation explicit, on-demand or implicit.

- **Explicit Activation**
  The server application explicitly activates individual objects using the `activate_object` or `activate_object_with_id` operations. This is most useful for server applications that have only a few objects.

- **On-demand Activation**
  The server application registers a servant manager (using `set_servant_manager`) that the POA upcalls when it receives a request for an object that is not activated.

- **Implicit Activation**
  If the POA has the IMPLICIT_ACTIVATION policy, as well as the RETAIN policy in effect, the POA can implicitly activate an object when the server application attempts to obtain a reference for a servant that is not already active. When you invoke the servant’s _this method, an action on the servant results in activation without any explicit calls on a POA.

These activation methods are described in the following subsections.

### 6.3.2 Explicitly Activating Objects

The method for explicitly activating an object depends on how IDs are assigned to objects managed by a POA.

- If the POA assigns the object IDs, you activate objects using the `activate_object` operation on the POA.
- If the application assigns the object IDs, you activate objects using the `activate_object_with_id` operation.

### 6.3.2.1 Using POA-assigned Object IDs

If a POA has the SYSTEM_ID policy in effect, objects are explicitly activated through the POA without providing a user-specified identity value. You call the `activate_object` operation on the POA that manages the object and the POA allocates, assigns and returns an object reference.

This capability is useful for transient objects, where the object ID needs to be valid only as long as the servant is active. The object IDs can remain completely hidden and no servant manager is needed. When POA-assigned object IDs are used with persistent objects or objects that are activated on demand, the application must be able to associate the generated object ID value with its corresponding object state.

**Examples**

Simple implementation of transient objects using POA-assigned object IDs. It presumes a POA that has the SYSTEM_ID, USE_SERVANT_MANAGER and RETAIN policies.

```java
// IDL
interface Foo
{
    long doit();
};
```
Derive the implementation from the generated **FooPOA** class.

```java
class MyFooServant extends FooPOA {
    MyFooServant( org.omg.PortableServer.POA poa, int value) {
        my_poa = poa;
        my_value = value;
    }
    public int doit(){ return my_value; }
    public org.omg.PortableServer.POA default_POA() {
        return my_poa; }
    private org.omg.PortableServer.POA my_poa;
    private int my_value;
}
```

Create the servant somewhere in the program during initialization, usually in `main()`.

```java
//create the servant
MyFooServant afoo = new MyFooServant (rootPOA, 42);
//explicitly activate the object
byte[] oid = poa.activate_object( afoo );
//acquire an object reference to the object
Foo foo = afoo._this();
//activate the POA, this also activates poa as they both
//have the same manager
rootPOA.the_POAManager().activate();
//export the new object reference to the trader
trading.register ( "MyFoo", "/", ",", foo); 
//run the ORB's event loop
orb.run();
```

### 6.3.2.2 Application-assigned Object ID

A server can explicitly activate an object by creating a POA that has the **USER_ID** policy and providing the application-assigned identity of that object. This capability is useful for objects that are used commonly or act as initial points of contact through which clients access other objects, for example, Factories. You can implement the server to create and explicitly activate these objects during initialization, avoiding the need for a Servant Manager.

The POA supports an explicit activation operation, `activate_object_with_id`, that associates a servant with an `ObjectId`. If the POA has the policy, **USE_SERVANT_MANAGER** and no Servant Manager is associated with a POA, any request that POA receives for an object ID value that is not present in the Active Object Map results in an **OBJECT_NOT_EXIST** exception.
Examples

Explicit activation of objects using application-assigned object IDs to activate all of the objects when the server is initialized. It assumes that a POA has the USER_ID,
USE_SERVANT_MANAGER and RETAIN policies. The code is the same as that shown for activation of objects with POA-assigned object IDs (see Section 6.1, Creating a POA), except for the code shown in **boldface** type.

```java
//create the servant
MyFooServant afoo = new MyFooServant (rootPOA, 42);
//create an object id
byte[] oid = (new String("myLittleFoo")).getBytes();
//explicitly activate the object using the created id
poa.activate_object_with_id( oid, afoo );
//acquire an object reference to the object
Foo foo = afoo._this();
//activate the POA, this also activates poa as they both
//have the same manager
rootPOA.the_POAManager().activate();
//export the new object reference to the trader
trading.register ( "MyFooServant", "/", "", foo);
//run the ORB's event loop
orb.run();
```

Create references for objects before activating them

```java
//create an object id
byte[] oid = (new String("myLittleFoo")).getBytes();
//create an object reference using the supplied id
org.omg.CORBA.Object obj = poa.create_reference_with_id
(oid, "IDL:Foo:1.0" );
//narrow to a Foo object type
Foo foo = FooHelper.narrow( obj );
//later, activate the object using the reference
//obtained earlier
MyFooServant afoo = new MyFooServant (rootPOA, 42);
poa.activate_object_with_id( oid, afoo );
```

6.3.3 Activating Objects on Demand

To activate objects on demand as they are needed, the server application must register a servant manager, using **set_servant_manager**, that the POA upcalls when it receives a request for an object that is not activated. The servant manager then performs one of these operations

- **Incarnates** the servant, if necessary, and registers it with the POA, which then dispatches the request to that servant.
- **Raises a ForwardRequest exception**, defined in the PortableServer module, to send the request to a different object.
- **Raises a CORBA::OBJECT_NOT_EXIST** exception to indicate that the object was destroyed.

If the servant manager raises a ForwardRequest exception, the orb2 ORB automatically resends the current request and all future requests to the object in the forward_reference member of the exception.

After activation, the POA maintains the association of the servant and the object ID in the Active Object Map if the POA has the **RETAINT** and **USE_SERVANT_MANAGER** policies.
Example
Transparent activation. The object ID value contains a key for a record in a database that contains the object’s state. The Servant Manager retrieves the state from the database, constructs a servant for implementation class, initializes it with the state from the database and returns it to the POA. The POA has the USER_ID, USE_SERVANT_MANAGER and RETAIN policies.

```java
//Create a foo object reference only. Note that the foo servant is not activated—this will be done in the ServantManager
byte[] oid = (new String("myLittleFoo")).getBytes();
org.omg.CORBA.Object obj = poa.create_reference_with_id
    ( oid, "IDL:Foo:1.0" );
Foo foo = FooHelper.narrow( obj );
//Now create the ServantManager servant.
MyFooServantActivator fooIM =
    new MyFooServantActivator();
//Obtain an object reference to the servant manager.
org.omg.PortableServer.ServantActivator IMref = fooIM._this();
//Associate the ServantManager with a poa.
poa.set_servant_manager(IMref);
rootPOA.the_POAManager().activate();
```

6.3.4 Activating Objects Implicitly

You can create a POA with a policy that implicitly activates its objects. This policy, IMPLICIT_ACTIVATION, also requires the SYSTEM_ID and RETAIN policies. When a POA supports implicit activation, you can implicitly activate an inactive servant in that POA by certain operations that require the assignment of an object ID to that servant.

Implicit activation of an object involves allocating a system-generated object ID and registering the servant with that object ID in the Active Object Map. The interface associated with the implicitly activated object is determined from the servant, using static information from the skeleton or, if it is a dynamic servant, using the _primary_interface() operation.

The operations and method that support implicit activation include

- **POA::servant_to_reference**
  Takes a servant parameter and returns a reference.

- **POA::servant_to_id**
  Takes a servant parameter and returns an object ID.

- **_this()** method
  Call from outside the context of an upcall to return an object reference for the servant.

If the POA has the UNIQUE_ID policy, implicit activation occurs when one of these operations is performed on a servant that is not currently active; that is, no association with an object ID in the POA's Active Object Map.

If the POA has the MULTIPLE_ID policy, the servant_to_reference and servant_to_id operations always perform implicit activation, even if the servant is already associated with an object ID.

Example
Use _this() operation to implicitly activate an inactive object
// Java
Demos.Calculator calc_ref = calc._this();

The _this() operation returns an object reference and implicitly activates the object if it is not already active. If you call _this() within the context of an upcall, it returns the object reference used to call the servant.

6.3.5 Activating Persistent Objects

Objects implemented in a POA that have the PERSISTENT policy can outlive the process in which they are first created. Persistent objects allow applications to exit and restart, while allowing clients to continue invoking requests on the object using its original object reference. Typically, the application also specifies the USER_ID policy for the POA so the application can define the object IDs.

When using persistent object, you may choose to support those objects inside a server that is manually launched. You must specify the listen port with either the -ORBport command-line option or you must use the com.twoab.orb2.port property.

Example

Generate a persistent object reference for a Calculator object. It gives the object reference to the Trader and exits.

/**
 * Program to generate a persistent object reference for a calculator object
 * File :: GenCalcServer.java
 *
 * First specify port 9000
 * Now start the ORB
 * Obtain object references for the Root POA and Trader.
 * Now create a child POA, beginning with specifying policies of PERSISTENT and USER_ID to allow persistent objects and application-assigned object IDs.
 * 
 * @param args
 */

import java.util.Properties;
import com.twoab.orb2.*;
public class GenCalcServer {
    public static void main(String[] args) {
        try {
            Properties defaultProperties = new Properties();
            // First specify port 9000
            defaultProperties.put("com.twoab.orb2.port", "9000");
            // Now start the ORB
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, defaultProperties);
            // Obtain object references for the Root POA and Trader.
            Trading trading = TradingHelper.narrow (orb.resolve_initial_references("orb2Trading"));
            // Now create a child POA, beginning with specifying policies of PERSISTENT and USER_ID to allow persistent objects and application-assigned object IDs.
            org.omg.CORBA.Policy[] policy_list = new org.omg.CORBA.Policy[2];
            policy_list[0] = poa.create_id_assignment_policy( //...
6.4 Activating the POA Manager

As described in the previous sections, each POA object has an associated POA Manager object. You can associate a POA Manager with one or more POA objects. A POA Manager encapsulates the processing state of the POAs it is associated with. Using operations on the POA Manager, an application can request to queue or discard and deactivate the POAs.

POA Managers are created and destroyed implicitly. Unless an explicit POA Manager object is provided at POA creation time, a POA Manager is created when a POA is created and is automatically associated with that POA. A POA Manager object is implicitly destroyed when all of its associated POAs are destroyed.

A POA Manager has these processing states: holding, active, inactive and discarding. The processing state determines the capabilities of the associated POAs and the disposition of requests received by those POAs. All POA Managers are created in the holding state. You must change the POA Manager from the holding state to the active state to dispatch requests. Invoke the `POAManager::activate` operation to perform this transition.

Table 6.5 describes what the POA does with requests, depending on the state of its POA Manager.

```java
   USER_ID );
policy_list[1] = poa.create_lifespan_policy(
   PERSISTENT );
org.omg.PortableServer.POA calcPOA =
   poa.create_POA( "calcPOA",
   poa.the_POAManager(),
   policy_list );
   // Now create an object id for the Calculator object
   byte[] calc_obj_id = { 1, 1, 2, 2 };
   // Create an object reference for the calculator
   // object using the object id created above and an
   // interface repository id.
   Demos.Calculator calc_ref =
   Demos.CalculatorHelper.narrow (calcPOA.create_reference_with_id(
   calc_obj_id,
   "IDL:Demos/Calculator:1.0" ) );
   // Register this object reference with the trader
   trading.register ("Calculator", "/", ",", calc_ref);
   // Note there is no orb.run(). This program exits
   // after a persistent object reference has been
   // registered.
}
}`

After the persistent object reference is registered with the Trader, any Calculator client can use that persistent object reference to contact the object; in this case, a Calculator object.
Activating the POA Manager

A POA Manager can be activated or deactivated by calling the `activate`, `discard_requests` or `deactivate` operations, respectively. You can issue the `get_state` operation to obtain the state of the POA Manager.

Table 6.5 POA Manager States

<table>
<thead>
<tr>
<th>POA Manager State</th>
<th>POA Processing Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>holding</td>
<td>Creates a POA Manager in this state, which causes the associated POAs to queue incoming requests. Adapter activators registered with the associated POAs are not called when the POA Manager is in this state.</td>
</tr>
<tr>
<td>active</td>
<td>The associated POAs receive and process requests.</td>
</tr>
<tr>
<td>inactive</td>
<td>The associated POAs reject new requests by returning an OBJ_ADAPTER exception to indicate that the object implementation is not available. The inactive state is entered using the deactivate operation when the associated POAs are to be shut down.</td>
</tr>
<tr>
<td>discarding</td>
<td>The associated POAs discard all incoming requests for which no processing has begun and returns the TRANSIENT system exception to the client to indicate to reissue the request. This state provides an application with flow-control capabilities when it determines that an object’s implementation or POA is being flooded with requests. After correcting the problem that caused the backlog of requests, the POA Manager can be restored to the active state.</td>
</tr>
</tbody>
</table>

An application changes a POA Manager from the holding state to either the active, discarding or inactive state by calling the `activate`, `discard_requests` or `deactivate` operations, respectively. You can issue the `get_state` operation to obtain the state of the POA Manager.

Figure 6.3 summarizes the operations that cause the POA Manager state to change. When the POA is deactivated, it is not possible to change the state. The POA is destroyed and you must recreate it.
The following is the IDL for the `POAManager` interface:

```java
interface POAManager {
    exception AdapterInactive();
    enum State {HOLDING, ACTIVE, DISCARDING, INACTIVE);
    void activate()
        raises(AdapterInactive);
    void hold_requests(
        in boolean wait_for_completion)
        raises(AdapterInactive);
    void discard_requests(
        in boolean wait_for_completion)
        raises(AdapterInactive);
    void deactivate(
        in boolean etherealize_objects,
        in boolean wait_for_completion)
        raises(AdapterInactive);
    State get_state();
};
```

When a POA Manager, including the Root POA, is created, it is in the holding state. Before a POA Manager can route requests to a servant, it must be activated.
6.5 Deactivating Objects

The following operation causes the ObjectId specified in the oid parameter to be deactivated, which removes it from the Active Object Map after all requests executing for that ObjectId have completed. If a Servant Manager is associated with the POA, ServantActivator::etherealize is invoked after the object ID is removed from the Active Object Map.

```java
// java
public void deactivate_object(in ObjectId oid)
    raises (ObjectNotActive, WrongPolicy;
```

You cannot reactivate the object until etherealization is complete. The deactivate_object operation returns immediately after deactivated the object ID, without waiting for requests or etherealization to complete.

**Note**  
Do not destroy a servant that is serving multiple object IDs. If the servant is associated with other objects when etherealize is called, the remaining_activations parameter returns "true." Do not destroy a servant unless the remaining_activations parameter is "false."

Example

Deactivate a POA

```java
// Java
public void destroy(in boolean etherealize_objects,
    in boolean wait_for_completion);
```

This operation destroys the POA and all descendent POAs, with the descendent POAs destroyed before the containing POA. When destroy is called, the POA does the following:

1. Calls destroy on all of its immediate descendants.
2. After all descendant POAs are destroyed and their servants etherealized, the POA continues to process requests until there are no requests executing in the POA.
3. If the etherealize_objects parameter is "true," the POA has the RETAIN policy, and a Servant Manager is registered with the POA, the etherealize operation on the Servant Manager is called for each active object in the Active Object Map.

**Note**  
If you call deactivate_object with the current thread in an invocation context dispatched from a POA that belongs to the same ORB as this POA, set the wait_for_completion parameter to "false." Setting the wait_for_completion parameter to "true" could cause a deadlock whereby the POA upcall function waits indefinitely for the destroy function to complete before it returns, and the destroy function waits indefinitely for all upcalls to complete. Spawn a thread in the upcall function and use it to perform the destroy operation so the upcall function can return.
Chapter 7

Portable Interceptors

This chapter provides a brief introduction to Portable Interceptors. It supports the OMG’s Portable Interceptor Specification defined in Chapter 21 of the CORBA Specification, Version 2.5.

7.1 Introduction

Portable interceptors are callback methods that are registered with the ORB by ORB Services. They are invoked by the ORB at certain points during the propagation of a message between the client and server.

- The invoked interception points are used by the ORB Service to query the request or reply parameters and modify the service contexts created by the server.
- An ORB Service may throw an exception at an interception to change or abort the flow of a request or reply between the client and the server.

There are two interceptor types:

- **request interceptors**: Intercepts the flow of the request or reply sequence during a method call.
- **IOR interceptors**: Adds extra information to object references passed between the client and the server.

7.2 Request Interceptors

An ORB service that wants to register request interceptors must first register an associated **ORBInitializer** object implementing the **ORBInitializer** interface. For the Java binding, it is necessary to specify the fully qualified name of the **ORBInitializer** classes file. The classes specified are dynamically loaded and instantiated during ORB initialization. The **ORBInitializer::pre_init** and **ORBInitializer::post_init** methods are invoked on the instantiated objects. The **pre_init** and **post_init** methods are responsible for registering the request interceptors with the ORB.

7.3 ORBInitializer Object

You must add the following property to the property file for each **ORBInitializer** class (typically one per ORB Service)

```conf
org.omg.PortableInterceptor.ORBInitializerClass.<classname>=<something>
```

where

- `<classname>`: Fully qualified class name of the service-specific **ORBInitializer** derived class.
- `<something>`: Arbitrary piece of text, for example, a textual description. This is present to ensure conformance with the Java *key=value* property file convention.
7.3.1 Registering Request Interceptors

During initialization, the ORB invokes the `pre_init` and `post_init` methods on all registered `ORBInitializer` objects, passing an instance of the `ORBInitInfo` class.

Each ORB service may register one or more request interceptor from within these methods. Request interceptors are registered as either client or server request interceptors using the `ORBInitInfo::add_client_request_interceptor` and `ORBInitInfo::add_server_request_interceptor` methods. These methods are passed an instance of a service-specific class implementing the `PortableInterceptor::ClientRequestInterceptor` or `PortableInterceptor::ServerRequestInterceptor` interfaces, respectively.

Example

```java
public class MyInitializer extends org.omg.CORBA.portable.ObjectImpl
    implements org.omg.PortableInterceptor.ORBInitializer {

    public void pre_init (org.omg.PortableInterceptor.ORBInitInfo info) {
    }

    public void post_init(org.omg.PortableInterceptor.ORBInitInfo info) {
        try {
            // Register the request interceptors ->
            info.add_client_request_interceptor(
                    new MyClientRequestInterceptor());
            info.add_server_request_interceptor(
                    new MyServerRequestInterceptor());
        } catch( Exception ex ) {
            System.err.println("Exception during initialization: "+ex);
            ex.printStackTrace();
        }
    }
}
```

Section 7.3.2 explains the difference between client and server request interceptors.

7.3.2 Ordering Request Interceptors

The OMG Portable Interceptors specification does not propose any mechanism that you can use to specify the order that request interceptors are invoked. That is, they are invoked in an arbitrary order, not necessarily in the order they are registered. This is potentially problematic since it is desirable to call the interceptors of some ORB services, for example, a security service, before any other interceptors are called. Without an ordering mechanism, this cannot be guaranteed. However, the orb2 for Java Portable Interceptors implementation supports an ordering mechanism for request interceptors, which avoids these limitations.

The order in which request interceptors are invoked is specified through two properties in the `orb.properties` file

```properties
com.twoab.orb2.client.request.interceptor.order=<list>
com.twoab.orb2.server.request.interceptor.order=<list>
```

where
7.4 Naming Interceptors

The interceptor name is specified by overriding the `Interceptor::name()` access method to return a string denoting the unique name of the interceptor. Although the name of the interceptor is not constrained to any particular pattern, we recommend following these guidelines:

- The name should reflect the name of the interception service. For example, for a tracing service `TraceInterceptor`
- The name may have the reverse-DNS domain name prefix of the company who developed the ORB service and, if applicable, the relevant product name of the company to distinguish it from another company’s similarly named service. For example `com.twoab.orb2.TraceInterceptor`
- The name must be unique.

7.5 Interception Points

There are several request interception points, which are invoked by the ORB during the course of the request/reply messages sent between the client and the server.

Client request interceptors define several client interception points. Server request interceptors define several side-side interception points. Table 7.1 summarizes the client and server request interception points.

<table>
<thead>
<tr>
<th>Client Interception Points</th>
<th>Server Interception Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>send_request</td>
<td>receive_request_service_context</td>
</tr>
<tr>
<td>send_poll</td>
<td>receive_request</td>
</tr>
<tr>
<td>receive_reply</td>
<td>send_reply</td>
</tr>
<tr>
<td>receive_exception</td>
<td>send_exception</td>
</tr>
<tr>
<td>receive_other</td>
<td>send_other</td>
</tr>
</tbody>
</table>

**Note** The `send_poll` interception point is not currently supported by orb2 for Java.
You can also classify interception points as starting, intermediate or ending interception points. This classification affects the order in which the interceptors are called, both in a normal message flow and in an exceptional message flow. Table 7.2 summarizes this classification.

Table 7.2 Starting/Intermediate/Ending Interception Points

<table>
<thead>
<tr>
<th>Starting Interception Points</th>
<th>Intermediate Interception Points</th>
<th>Ending Interception Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>send_request</td>
<td>receive_request</td>
<td>send_reply</td>
</tr>
<tr>
<td>send_poll</td>
<td></td>
<td>receive_reply</td>
</tr>
<tr>
<td>receive_request_service_context</td>
<td></td>
<td>receive_exception</td>
</tr>
<tr>
<td></td>
<td></td>
<td>receive_other</td>
</tr>
<tr>
<td></td>
<td></td>
<td>send_exception</td>
</tr>
<tr>
<td></td>
<td></td>
<td>send_other</td>
</tr>
</tbody>
</table>

Note  The send_poll interception point is not currently supported by orb2 for Java.

The following summarizes the rules for invoking request interceptors. You should also refer to the OMG specification for a detailed discussion of request interceptors and the order in which the interception points are invoked by the ORB. This document also gives a number of use-case scenarios.

- During the progression of a message from the client to the server (and vice versa), one or more starting interception points are called. For the server, this is receive_request_service_context and for the client, this is send_request and/or send_poll.
- If a starting interception point throws an exception, the remaining uncalled starting interception points are not invoked.
- For each invoked starting interception point, a corresponding ending interception point is always invoked.
  - System exception
    Server   send_reply
    Client   receive_reply
  - Forward request exception
    Server   send_other
    Client   receive_other
- If a starting interception point throws an exception, the ORB invokes an ending interception point for each starting interception point that was invoked so far.
  Server   send_exception
  Client   receive_exception
- During normal message flow, for the server, the receive_request intermediate interception point is invoked after receive_request_service_context and before the send_reply ending interception point is invoked for the server.
  - If the receive_request_service_context interception point raises a system exception, the receive_request interception point is not invoked and the send_exception ending interception point is invoked for all receive_request_service_context interception points invoked so far.
  - If the receive_request_service_context interception point raises a forward request exception, the receive_request interception point is not invoked and the send_other ending interception point is invoked for all receive_request_service_context interception points invoked so far.
7.6 IOR Interceptor Points

The IOR interception points are invoked by the ORB when it is assembling the list of tagged components that go into an IOR. The invoked interception points may use the IORInfo interface to add additional components to the list. The additional components are used to associate some service-specific attributes with an object reference.

Example

Associate an object reference with a specific POA policy. This is used by an ORB service to determine which objects adhere to a certain policy.

An object reference is associated with a certain transactional policy so that it is identified by a transaction service.

An object reference is associated with a certain security policy so that it is identified by a security service.

7.6.1 Registering IOR Interceptors

During initialization, the ORB invokes the ORBInitializer::pre_init and ORBInitializer::post_init methods on all registered ORBInitializer objects, passing an instance of the ORBInitInfo class.

ORB Services that want to add one or more tagged components to an IOR must first register an instance of an object, implementing the IORInterceptor interface by invoking the ORBInitInfo::add_ior_interceptor method.

Example

```java
    public void pre_init (org.omg.PortableInterceptor.ORBInitInfo info) {
    }
    public void post_init(org.omg.PortableInterceptor.ORBInitInfo info) {
        try {
            // Register an IOR interceptor ->
            info.add_ior_interceptor(
                new MyIORInterceptor());
         }
        catch( Exception ex )
        {
            System.err.println("Exception during initialization: " + ex);
            ex.printStackTrace();
        }
    }
}
```

7.7 Invoking IOR interceptors

The IORInterceptor.establish_components method is invoked for each registered IOR interceptor at the time the ORB is assembling the list of tagged components for inclusion in an IOR.

Note In orb2 for Java, the list of tagged components is only assembled once. Therefore, the IORInterceptor.establish_components method is only executed once on each registered interceptor.
The body of the ORB Service’s `establish_components` method invokes the `add_ior_component` or `add_ior_component_to_profile` methods to add a tagged component to the set included in an IOR.

Refer to the OMG Portable Interceptors specification for further details on using IOR Interceptors.

## 7.8 Tracing Facility Example

This section provides a step-by-step example showing an implementation for a tracing facility using portable interceptors. The example does not demonstrate every possible function of the Portable Interceptors mechanism. However, the most useful methods are demonstrated. Refer to the OMG Portable Interceptors specification for guidance on the remaining functionality.

This example is for a client-server application in which the client invokes a single method, `increment`, on the servant. This method increments the integer value passed to it by 1 and returns the result. The following IDL is used to define the interface for the example application.

```idl
interface InterceptorsExample
{
    // Method declarations:
    // Test method:
    long increment(in long value);
    // Kill thyself:
    void kill();
};
```

The Portable Interceptors mechanism is used to add tracing to the request/reply message flow between the client and the server by printing out a message to the console each time an interception point is invoked. The trace messages are printed to the console via a simple tracing class; `Trace`. You can refine it to provide a more useful tracing/logging facility (for example, auditing) by redirecting the output to a file instead of the screen.

### 7.8.1 Registration

The following property is added to the `orb.properties` file for the initialization class (`Initializer`). The initialization class implements the `ORBInitializer` interface.

```
org.omg.PortableInterceptor.ORBInitializerClass.Initializer=Interceptors
```

### 7.8.2 Initialization Class

The `Initializer` class is responsible for registering the request interceptors with the ORB. The `post_init` method invokes the `add_client_request_interceptor` and `info.add_server_request_interceptor` methods to add an instance of the example’s client and server request interceptor classes (`ClientInterceptor` and `ServerInterceptor`).

```java
public class Initializer
    extends org.omg.CORBA.portable.ObjectImpl
    implements org.omg.PortableInterceptor.ORBInitializer {

    public void pre_init (org.omg.PortableInterceptor.ORBInitInfo info) {
    }

g public void post_init(org.omg.PortableInterceptor.ORBInitInfo info) {
    try {
        // Register the request interceptors for the example
        info.add_client_request_interceptor(new ClientInterceptor());
        info.add_server_request_interceptor(new ServerInterceptor());
    } finally {
```
catch( Exception ex )
{
    System.err.println("Exception during post_init: " + ex);
    ex.printStackTrace();
}
}

7.8.3 Request Interceptor Classes

The **ClientInterceptor** class implements the **ClientRequestInterceptor** interface and contains the implementation for the client interception points. Similarly, the **ServerInterceptor** class implements the **ServerRequestInterceptor** interface and contains the implementation for the server interception points. Also note the following in this example:

- The interception points are invoked at the appropriate junctures in the flow of the request/reply message from the client to the server (and vice versa).
- Each interception point invokes the **println** method in the **Trace** class to produce a trace message. The **Trace** class is a singleton object and the static method **instance** is used to create this object and return a reference to it.
- The **name** method is overridden to return a name for the interceptor. This is important if at some stage the interceptor needs to be ordered.
- The **ClientInterceptor.send_poll** method is never invoked. orb2 for Java does not currently support this interception point.

```java
public class ClientInterceptor
    extends org.omg.CORBA.portable.ObjectImpl
    implements ClientRequestInterceptor
{
    public ClientInterceptor()
    {
    }

    public java.lang.String name( )
    {
        return "com.twoab.orb2.ExampleClientInterceptor";
    }

    public void send_request(ClientRequestInfo ri)
        throws org.omg.PortableInterceptor.ForwardRequest
    {
        Trace.instance().println(Trace.TRACE_LEVEL_INFO,
            "ExampleClientInterceptor: send_request invoked.");
    }

    public void send_poll(ClientRequestInfo ri)
    {
        // no op at the moment
    }

    public void receive_reply(ClientRequestInfo ri)
    {
        Trace.instance().println(Trace.TRACE_LEVEL_INFO,
            "ExampleClientInterceptor: receive_reply invoked.");
    }

    public void receive_exception(ClientRequestInfo ri)
        throws org.omg.PortableInterceptor.ForwardRequest
    {
        Trace.instance().println(Trace.TRACE_LEVEL_INFO,
            "ExampleClientInterceptor: receive_exception invoked.");
    }

    public void receive_other(ClientRequestInfo ri)
        throws org.omg.PortableInterceptor.ForwardRequest
    {
        Trace.instance().println(Trace.TRACE_LEVEL_INFO,
            "ExampleClientInterceptor: receive_other invoked.");
    }

    public class ServerInterceptor
    {
    }
}
```
extends org.omg.CORBA.portable.ObjectImpl
implements ServerRequestInterceptor
{
    public ServerInterceptor()
    {
    }
    public java.lang.String name()
    {
        return "com.twoab.orb2.ExampleServerInterceptor";
    }
    public void receive_request_service_contexts (ServerRequestInfo ri)
        throws org.omg.PortableInterceptor.ForwardRequest
    {
        Trace.instance().println(Trace.TRACE_LEVEL_INFO,
            "receive_request_service_contexts invoked.");
    }
    public void receive_request (ServerRequestInfo ri)
        throws org.omg.PortableInterceptor.ForwardRequest
    {
        Trace.instance().println(Trace.TRACE_LEVEL_INFO,
            "receive_request invoked.");
    }
    public void send_reply (ServerRequestInfo ri)
    {
        Trace.instance().println(Trace.TRACE_LEVEL_INFO,
            "send_reply invoked.");
    }
    public void send_exception (ServerRequestInfo ri)
        throws org.omg.PortableInterceptor.ForwardRequest
    {
        Trace.instance().println(Trace.TRACE_LEVEL_INFO,
            "send_exception invoked.");
    }
    public void send_other (ServerRequestInfo ri)
        throws org.omg.PortableInterceptor.ForwardRequest
    {
        Trace.instance().println(Trace.TRACE_LEVEL_INFO,
            "send_other invoked.");
    }
}

7.8.4 Killing the Server

The KillServer class is a simple client application that invokes the kill method on the server to unregister the interceptors example object with the Trader and spawns a separate thread (by creating an instance of the ServerShutdown class) to perform a tidy shutdown of the server and the ORB.

This class is provided for the user as an alternative to CTRL+C to stop the server. Unlike CTRL+C, it does not result in stale offers in the Trader database.

7.8.5 Server Class

The Server class is the server application that performs the following
• Initializes the ORB
• Obtains an object reference to the RootPOA and registers a new child POA with a an Implicit Activation policy
• Uses the new child POA to create a servant
• Activates the child POA and exports the servant object reference to the Trader
• Starts the ORB’s event loop to listen for and service incoming client requests

Refer to the other sections of this document for details about initializing the ORB, creating POAs and exporting object references to the Trader.
The **Server** class contains an inner class, **InterceptorsExampleServant**, which contains the implementation for the **increment** and **kill** methods of the servant interface. Review the source code of the interceptors example for the implementation of these methods.

### 7.8.6 Client Class

The **Client** class is the client application that performs the following:

- Analyzes the command-line arguments and extracts the value to send to the server application to be incremented
- Initializes the ORB
- Locates the servant object via the Trader
- Invokes the **increment** method on the servant
- Displays the result
- Shuts down the ORB and exits the client application

Review the source code of the interceptors example for further details.
Chapter 8  DII/DSI

This chapter describes how the Dynamic Invocation Interface (DII) and the Dynamic Skeleton Interface (DSI) allows you to add flexibility to distributed applications. Complete information about how to use Dynamic Interfaces can be found in the OMG's CORBA specification.

8.1 Static Requests
When using static request invocations, you use the stub and skeleton classes that are generated by an IDL compiler. Client applications use the stub classes to invoke operation on remote servers. Servers use the skeleton classes to receive the incoming invocations. This is the typical means of building distributed applications with CORBA.

8.2 Dynamic Requests
There are some applications that may not have knowledge of the services that it might invoke until runtime. That is, when the client application is written and compiled, it is not aware of all the servers that it might need to use. An example of such an application might be a testing tool. Such an application would not have any prior knowledge of the workings of the servers that it might test. It must provide a means of obtaining that information at runtime (e.g. from an operator).

Another type of specialized application would be a server that is not aware of the operations that it must support until runtime. An example of such a server might be a server that does load balancing and redirects operations to servers that are not currently busy.

In the case of the client application described above, it could use the Dynamic Invocation Interface (DII). It would not require pre-compiled stub files. The server described above could use the Dynamic Skeleton Interface (DSI) and would not require pre-compiled skeleton files.

8.3 Invoking Dynamic Requests
The Dynamic Invocation Interface (DII) allows the dynamic construction of a request, without the need for a pre-compiled stub class. It allows the dynamic construction of an object invocation. The client application must supply information about the operation and the type and value of parameters that are to be passed with the operation. Objects that receive these invocations will have no way of knowing if the operation was invoked statically or dynamically.

8.4 Receiving Requests Dynamically
The Dynamic Skeleton interface (DSI) provides a way to deliver requests from an ORB to an object implementation that does not have compile-time knowledge of the object it is implementing. The DSI implements all requests on a particular object by having the ORB the same upcall routine, a Dynamic Implementation Routine (DIR). You may use a single DIR as the implementation for many objects with different interfaces.

8.5 Dynamic Any's
When dynamically creating and receiving requests, parameters are inserted/extracted from CORBA Any types. One of the problems of invoking operations upon objects that one has no prior knowledge about is that one probably also does not have any prior knowledge about the
data types that may be passed as parameters, especially user defined data types. IDL compilers create classes that define user-defined data types and provide helper classes that support inserting and extracting these data types into and out of CORBA Any types. Without these classes, it becomes necessary to use the CORBA Dynamic Any mechanism to construct these values.

8.6 Performance Issues

Often, people assume that by using the DII and/or DSI interfaces, they might achieve some performance enhancements. They make this assumption based on the fact that they are bypassing code found in stub and skeleton code. Typically, this assumption is false. DII and DSI code must perform the same functionality that the pre-compiled stub and skeleton code, and it may also have the burden of not having prior knowledge of things such as data types.

The DII and DSI interface should be used only for those application that truly have the requirement that they have no prior knowledge of the objects that they may wish to use.

8.7 Demonstration Programs

DII and DSI usage is illustrated in the demonstration programs in the dynamic subdirectory of the demos directory. In this example, a server offers a simple facility to clients for reversing strings. The revers.idl file describes the implementation for the string reversing functions.

Refer to the files in this directory for a detailed comparison of the static and dynamic demonstration programs.
This chapter describes the use of the RMI-IIOP protocol.

9.1 Introduction to RMI-IIOP

The initial releases of the Java environment supported a form of distributed object computing known as Remote Method Invocation (RMI). Although it was neither as robust or efficient as CORBA, it did possess the very useful capability of passing Java objects as parameters and return values of remote invocations. RMI did possess a major drawback. It used a proprietary network protocol and could only be used with Java clients and servers. It could not interoperate with new or existing services written in other languages such as C, C++, etc.

Java 2, version 1.3, introduced RMI-IIOP. This version used the OMG’s IIOP protocol as the standard for interoperability and supported the Java Language Mapping to OMG IDL Specification to support interoperability with other programming languages. The Java RMI interface is mapped to OMG IDL that can be implemented on a CORBA compliant ORB using the language binding of choice. In this scenario, Java objects are mapped to IDL value types.

A good starting point for programmers wanting to learn how to use RMI-IIOP is the URL http://java.sun.com/products/rmi-iiop/. In addition, there are numerous books that provide tutorials for developers using RMI-IIOP.

The definitive document for specifying the use of RMI-IIOP is the OMG specification titled Java to IDL Mapping Specification (formal/2002-08-06).

9.2 Using RMI-IIOP with orb2 for Java

There are two basic ways to use the RMI-IIOP protocol with orb2 for Java. The first technique is to simply use the RMI-IIOP protocol for both the client and server as described in Java’s RMI-IIOP documentation. This technique requires the use of Java JDK 1.4 or higher. The second technique is to write a client or server using pure CORBA programming techniques that can communicate with an RMI-IIOP-based client or server. The second technique would employ the use of value types for passing Java objects and could prove to be a more complex task than using the first technique. Of course, if a different ORB for another language was used, the second technique would be the only available method to interoperate with an RMI-IIOP program. This technique may be used with Java JDK 1.3 or higher.

There are two demonstration programs in the demos/rmi subdirectory of the orb2 for Java installation. The first demo illustrates the use of RMI-IIOP for both the client and server. The second demo illustrates the use of a pure CORBA client communicating with an RMI-IIOP server. Both demonstrations use POA-based RMI servers and thus require the use of JDK 1.4 or higher. The readme.txt files located with the demos explain how to build and run the programs.

9.3 Special Codeset Requirements

When developing a client application (using RMI-IIOP) that will interoperate with the ORB provided by JDK 1.3, you must specify special codesets for both regular and wide characters. This is done by setting the com.twoab.orb2.char_set property to a value of ISO_646 and the com.twoab.orb2.wchar_set property to a value of UCS_2.

Note This should ONLY be used when interoperating with the ORB provided by JDK 1.3.
9.4 RMI/IDL Subset

An extensive subset of Java RMI data types can be mapped to IDL types and hence passes as parameters and/or return values. The following subsections describe the supported Java data types.

9.4.1 Primitive Types

Standard Java primitive types are supported. They include void, boolean, byte, char, short, int, long, float and double.

9.4.2 RMI Remote Interfaces

This can be any interface that inherits from java.rmi.Remote either directly or indirectly and is defined to throw java.rmi.RemoteException.

9.4.3 RMI Value Types

This is any Java object that implements either java.io.Serializable or java.io.Externalizable. This type must not also implement java.rmi.Remote.

9.4.4 Arrays

This can be any array of supported Java data types.

9.4.5 Exceptions

Exceptions are supported if 1) it is a checked exception class and 2) it is a valid RMI Value Type.

9.4.6 CORBA Object References

This object must extend org.omg.CORBA.Object directly or indirectly.

9.4.7 IDL Entity Types

This object must extend org.omg.CORBA.portable.IDLEntity. Examples are classes generated from IDL for structs, enums, etc.

9.5 Building an RMI-IIOP Server Program

Building a server program is very similar to building a standard CORBA server. The following subsections describe the steps required to build an RMI-IIOP server.

9.5.1 Define the interface

You must define the interface class that defines the remote operations supported by the server. This interface must extend java.rmi.Remote. The following code is an example:

```java
public interface RmiInterface extends java.rmi.Remote {
    public int echoInteger(int val) throws java.rmi.RemoteException;
}
```
9.5.2 Implement the interface

You must provide an implementation of the interface defined above. This implementation must extend `javax.rmi.PortableRemoteObject`. The following code is an example:

```java
class RmiImpl extends javax.rmi.PortableRemoteObject {
    public RmiImpl() throws java.rmi.RemoteException {
        super();
    }

    public int echoInteger(int val) {
        return val;
    }
}
```

9.5.3 Generate Skeleton Classes

Just as with normal CORBA server, RMI-IIOP server use skeleton classes (clients us stub classes). With normal CORBA, an IDL compiler is used to generate the skeleton classes. With RMI-IIOP, the Java "rmic" compiler is used to generate skeleton classes. This is done after the implementation class is compiled. In the example described in this section, the command would be:

```
rmic -iiop -nolocalstubs -poa RmiImpl
```

If it is desired to generate IDL for use by pure CORBA clients, also use the "-idl" command-line argument.

9.5.4 Implement the Server

Implementing the server class is almost identical to implementing a pure CORBA server. You must initialize an ORB, POA and server implementation and then run the ORB. You also make the object reference available to clients though some service such as the Naming Service (This is not illustrated in the following code). The following code is an example:

```java
// Set properties for orb2
Properties props = System.getProperties();
props.setProperty("org.omg.CORBA.ORBClass", "com.twoab.orb2.core.ORBImpl");
props.setProperty("org.omg.CORBA.ORBSingletonClass", "com.twoab.orb2.core.ORBSingleton");
ORB orb = ORB.init(args, props);

// Initialize POA
org.omg.PortableServer.POA root_poa = orb.resolve_initial_references("RootPOA");
org.omg.PortableServer.POAManager mger = root_poa.the_POAManager();
org.omg.CORBA.Policy [] policies = new org.omg.CORBA.Policy[1];
policies[0] = root_poa.create_lifespan_policy(LifespanPolicyValue.PERSISTENT);
org.omg.PortableServer.POA my_poa = root_poa.create_POA("MyPoa", null, policies);
```
9.6 Building an RMI-IIOP Client Program

Building a client program is very similar to building a standard CORBA client. The following subsections describe the steps required to build an RMI-IIOP client.

9.6.1 Locate Object Reference

The client program must initialize an ORB and locate an object reference for a service in the manner prescribed by the service. This will typically be by using the Naming Service, Trader, etc. The following code is an example:

```java
my_poa.the_POAManager().activate();

// Initialize implementation
RmiImpl the_impl = new RmiImpl();
_RmiImpl_Tie tie = (_RmiImpl_Tie)javax.rmi.CORBA.Util.getTie(impl);
String obj_id = "MyRmi001";
my_poa.activate_object_with_id(obj_id.getBytes(), tie);
org.omg.CORBA.Object objref =
my_poa.create_reference_with_id(obj_id.getBytes(),
tie._all_interfaces(my_poa, obj_id.getBytes())[0]);

// Register object reference with some location service (e.g. Naming)
... code not shown

orb.run();
```

```java
// Set properties for orb2
Properties props = System.getProperties();
props.setProperty("org.omg.CORBA.ORBClass",
"com.twoab.orb2.core.ORBImpl");
props.setProperty("org.omg.CORBA.ORBSingletonClass",
"com.twoab.orb2.core.ORBSingleton");
ORB orb = ORB.init(args, props);

// Obtain the Object Reference - technique is not shown since it is
location service dependent
java.lang.Object obj = ....

// Narrow the reference
RmiInterface ref =
(RmiInterface)java.rmi.PortableRemoteObject.narrow(obj,
RmiInterface.class)
```
9.6.2 Invoking Operations

Once the object reference has been obtained, all that remains is to use that reference to invoke operations on the remote service. The following code is an example:

```java
// The reference may now be used to invoke operations
try {
    int ret_val = ref.echoInteger(123);
} catch (Exception e) { ... }
```

9.7 RMI Codebase

The orb2 for Java’s implementation of RMI-IIOP supports the RMI notion of a remote codebase. That is, a client may remotely load classes needed to communicate with an RMI-IIOP server. This can include stub classes, etc.

Clients must establish a security manager (e.g. java.rmi.RMISecurityManager), have permissions for the security manager and set the `java.rmi.server.codebase` property to define a URL for an HTTP or FTP server for remote classes.

Please see Java RMI documentation for complete details regarding using a remote codebase.

9.8 CORBA Clients and Servers

Thus far, this chapter has discussed the use of RMI-IIOP for both the client and server application code; however, there might be a scenario where the client and server use different techniques. One would use CORBA-IIOP and the other RMI-IIOP. The following sections briefly discuss the techniques that would be used for these mixed development environments.

9.8.1 CORBA-IIOP Client and RMI-IIOP Server

In this scenario, one has an existing RMI-IIOP server and wants to develop a client application, using CORBA-IIOP, that uses the server. This can be simply done by using the IDL generated by the development process for the server. The client development process simply uses that IDL to develop a CORBA-IIOP client just as it would for any other CORBA service.

**Note** This technique, although perfectly valid, is more likely to be used in development environments using a language other than Java (e.g. C++, C). The use of value types to map the Java objects typically adds more complexity than required for the Java environment.

9.8.2 RMI-IIOP Client and CORBA-IIOP Service

In this scenario, one has an existing CORBA-IIOP server and wants to develop a client application, using RMI-IIOP, that uses the server. In this scenario, the client application will need access to RMI stub classes that provide access to the CORBA service. Unfortunately, there is no program that takes CORBA IDL and produces RMI stubs. It is thus necessary to manually create an interface class (that maps the CORBA IDL operations) and an implementation class (actual implementation is not important). These can now be used by the Java "rmic" compiler to generate RMI stubs for the client.

**Note** This technique, although perfectly valid, probably introduces more complexity than writing the client using CORBA-IIOP and probably should be used only by existing clients that are already using RMI-IIOP and do not want any inconsistent techniques used to access services.
The orb2 for Java product provides support for the CORBA/IIOP specification. As such, it supports the class interfaces as defined in the specification. Similarly, the product supports the CORBA Naming Service, whose interfaces are defined in the Naming Service specification.

The purpose of this chapter is to provide a reference for those interfaces that are supported in the product that are an extension to OMG specifications. All of the classes described in this chapter are found on the `orb2.jar` file.

### 10.1 com.twoab.orb2.util.IORFormat

**Description**

The `IORFormat` class is used to format an object reference and/or to extract selected information from an object reference.

**Constructors**

- `public IORFormat(org.omg.CORBA.Object obj, org.omg.CORBA.ORB orb)`
  
  This form of constructor takes the object reference to be formatted, `obj`, and a reference to an initialized ORB, `orb`. It can throw an `IORFormatException` exception.

- `public IORFormat(org.omg.CORBA.Object obj)`
  
  This form of constructor takes the object reference to be formatted, `obj`. It can throw an `IORFormatException` exception.

- `public IORFormat(String ior_str)`
  
  This form of constructor takes the stringified representation, `ior_str`, of the object reference to be formatted. It can throw an `IORFormatException` exception.

**Public Methods**

- `public String getHost()`
  
  Returns the host name found in the object reference.

- `public int getPort()`
  
  Returns the port number found in the object reference.

- `public String getType()`
  
  Returns the repository ID for the target represented by the object reference.

- `public byte getIIOPMajorVersion()`
  
  Returns a byte containing the IIOP major version represented by this object reference.

- `public byte getIIOPMinorVersion()`
  
  Returns a byte containing the IIOP minor version represented by this object reference.
public String getIIOPVersion
    Returns a string that contains the IIOP major and minor version represented by this object reference.

public byte[] getObjectKey()
    Returns the object key found in the object reference.

public String getObjectKeyAsString()
    Returns a stringified representation of the object key found in the object reference.

public TaggedProfileFormat[] getTaggedProfiles()
    Returns an array of TaggedProfileFormat objects. Each object in the returned array represents a Tagged Profile found in the object reference.
10.2 com.twoab.orb2.util.IORFormatException

Description
The IORFormatException class is used to indicate an error condition when formatting contents of an object reference.

Constructors
public IORFormatException()
10.3  com.twoab.orb2.util.IORPrint

Description

The IORPrint class is used to display contents of a stringified object reference on a system console. In addition to details available via the IORFormat class, it can print details for the CSI_SEC_MECH_LIST tagged component (used with CSIv2) and the CODE_SETS tagged component (used with wchar and wstring).

Constructors

public IORPrint(String file_path, int options)

This form of constructor takes the full path name of a file containing a stringified IOR and an integer represent various options. No options are supported at this time.

public IORPrint(String ior_str)

This form of constructor takes a single parameter, a stringified IOR.

Public Methods

public static void main(String[] args)

This method is for running a program to produce IOR information to a console. It takes a single command-line argument that is the full path of a file containing a stringified IOR.

public void print()

This method displays the details of the IOR on the console.
10.4  com.twoab.orb2.util.ProfileFormat

Description

The ProfileFormat class is extract information from a TAG INTERNET_PROFILE profile. ProfileFormat objects are obtained from the getProfile method of the TaggedProfileFormat class.

Constructor

ProfileFormat objects are obtained via the getProfile method of the TaggedProfileFormat class.

Public Methods

public String getHost()
    Returns the host name found in the object reference.

public int getPort()
    Returns the port number found in the object reference.

public String getType()
    Returns the repository ID for the target represented by the object reference.

public byte getIIOPMajorVersion()
    Returns a byte containing the IIOP major version represented by this object reference.

public byte getIIOPMinorVersion()
    Returns a byte containing the IIOP minor version represented by this object reference.

public String getIIOPVersion
    Returns a string that contains the IIOP major and minor version represented by this object reference.

public byte[][] getObjectKey()
    Returns the object key found in the object reference.

public String getObjectKeyAsString()
    Returns a stringified representation of the object key found in the object reference.

public TaggedComponentFormat[] getComponents()
    Returns an array of TaggedComponentFormat objects. Each object in the returned array represents a Tagged Component found in the object reference.
10.5  com.twoab.orb2.util.SystemExceptionFormat

Description
The SystemExceptionFormat class is used to extract information from a CORBA System Exception.

Constructors
public SystemExceptionFormat(org.omg.CORBA.SystemException ex)
This form of constructor takes a single parameter, a CORBA System Exception.

Public Methods
public static String format(org.omg.CORBA.SystemException ex)
This method takes a single parameter, a CORBA System Exception and returns a string with formatted information about the exception.

public static String format(org.omg.CORBA.SystemException ex, boolean status, boolean code, boolean reason)
This method takes a parameter representing the CORBA System Exception to format and three Boolean values that indicates what information to include. If the status parameter has a value of True, completion status information will be included. If the code parameter has a value of True, the minor code value will be included. If the reason parameter has a value of True, the string describing the exception reason will be included.

public String getCompletionStatus()
Returns a string representing the completion status of an invocation.

public String getMinorCode()
Returns a string representing the exception’s minor code.

public String getReason()
Returns a string describing the reason for the exception.

public String getType()
Returns a string name of the exception that was thrown.
10.6 com.twoab.orb2.util.TaggedComponentFormat

Description
The TaggedComponentFormat class is used to provide information about a tagged component found within a profile.

Constructor
TaggedComponentFormat objects are obtained via the getComponents method of the ProfileFormat class.

Public Methods
public String getComponentAsString()
    Returns a representation of the tagged component.

public int getTag()
    Returns the integer that identifies the component (as defined in the CORBA specification).

public String getTagAsString()
    Returns a string that matches the name for the component (as defined in the CORBA specification).
10.7 com.twoab.orb2.util.TaggedProfileFormat

**Description**

The `TaggedProfileFormat` class is used to provide information about a tagged profile found within an object reference.

**Constructor**

`TaggedProfileFormat` objects are obtained via the `getTaggedProfiles` method of the `IORFormat` class.

**Public Methods**

- `public ProfileFormat getProfile()`
  
  Returns a `ProfileFormat` object that represents the profile.

- `public int getTag()`
  
  Returns the integer that identifies the profile (as defined in the CORBA specification).

- `public String getTagAsString()`
  
  Returns a string that matches the type of profile (as defined in the CORBA specification).
10.8  com.twoab.orb2.Trader

Description

The TraderOperations class implements the Trader. The Trader matches offers to support a specific service with requests for that service. A service provider indicates that it can provide a service by registering a service offer with the Trader. A service consumer requests that the Trader provide it with at least one service offer that provides the service it needs.

Constructors

Use the ORB’s resolve_initial_references method with the orb2_Trader argument to obtain a Trader object.

Public Methods

void register (java.lang.String interface_name,
               java.lang.String naming_context,
               java.lang.String prop_list,
               org.omg.CORBA.Object obj);

This method places a new service offer in the Trader's database. The new offer is associated with interface_name as its interface type. The naming_context is the context name under which the offer is stored, and the property names and values contained in prop_list are associated with the offer, as is the object reference obj. If the operation is unsuccessful, an appropriate exception is raised.

void lookup (java.lang.String interface_name,
            java.lang.String naming_context,
            java.lang.String constraints,
            TraderPackage.LookUpPolicy policy,
            TraderPackage.OffersHolder results);

This method searches the Trader database for a matching offer according to the specified policy. The user specifies the required type with interface_name, the root of the context tree in which to search using naming_context and a set of matching constraints for the offers property values through constraints. Three lookup policies are currently supported:

- **lookup_random**
  If specified and more than one matching offer is found, one is chosen at random by the Trader for return to the requester.

- **lookup_all**
  If specified, returns all matching offers. If the lookup fails, it raises an exception. If it succeeds, returns a sequence of the Offer structure in result.

- **lookup_proxy_all**
  This is similar to lookup_all except that if the offer is a proxy, the call returns the proxy instead of forwarding it. This avoids inadvertent instantiation of Managed Objects that are registered as proxies.

void delete (org.omg.CORBA.Object obj, java.lang.String constraints);

This method deletes all offers in the Trader database that have the specified object reference (obj) and match the constraints supplied in constraints. If the operation is unsuccessful, raises an appropriate exception.
10.9  com.twoab.orb2.Trading

Description

The Trading interface provides similar facilities to the Trader interface but with more resilience. Instead of being associated with a specific remote Trader, a Trading object maintains references to both the Primary Trader and, if configured, the Secondary Trader. If a Trading operation is invoked when the Primary Trader is unavailable, the operation is automatically redirected to the Secondary Trader. Using the Trader interface, an exception is raised and the program must obtain an object reference for the Secondary Trader and retry the operation. Because of this extra resilience, you should use Trading instead of Trader.

Refer to the orb2 Trader Service User's Guide for a description of Secondary Trading. Refer to the Trading example for typical uses of the Trading object.

Inheritance

This interface inherits from the Trader interface. The methods inherited from the Trader interface are also available, such as lookup(), register() and delete(), and if invoked on a Trading object, Trading can use these features.

Constructors

Use the ORB’s resolve_initial_references method with an argument of orb2Trading to obtain a Trading object.

Public Methods

org.omg.CORBA.Object _import (java.lang.String interface_type, java.lang.String interface_context, java.lang.String interface_props);

This method is similar to the Trader.lookup() operation. It causes the Trader to search its database for an offer matching the specified type, context and properties. If an offer is found, it is returned using the return value. If several matching offers are found, one is selected at random. If the operation is unsuccessful, an appropriate exception is raised.

void export (org.omg.CORBA.Object obj_ref, java.lang.String interface_type, java.lang.String object_context, java.lang.String object_props);

This method is similar to the Trader.register() operation. It places the obj_ref object in the offer database of the Trader with the specified type, under the specified context and with the specified property list. If the operation is unsuccessful, an appropriate exception is raised.

void withdraw (org.omg.CORBA.Object obj_ref);

This method is similar to the Trader.delete() operation. It removes any offers for the obj_ref object from the Traders database. If the operation is unsuccessful, an appropriate exception is raised.

void withdraw_constraint (org.omg.CORBA.Object obj_ref, java.lang.String constraints);

This method is similar to the Trader.delete() operation. It removes any offers for the obj_ref object that match the specified constraints from the Traders database. Offers for obj_ref that do not match the constraints remain in the database. If the operation is unsuccessful, an appropriate exception is raised.
void unexport (org.omg.CORBA.Object obj_ref);

This method is an asynchronous version of the `withdraw` method. A request to remove offers associated with `obj_ref` is sent to the Trader, but it does not wait for a response. Therefore, errors are not reported.
Symbols

_alls_interfaces() 2-5
__primary_interface 6-16
__this 6-8, 6-13
__this method 6-16
__this_object 2-9
__this_object() 2-6
__this_object(ORB orb) 2-5
__this_object(orb) 2-7

A

a_POAManager 6-5
activate 6-10
activate operation
POA Manager 6-19
activate_object 6-13
activate_object_with_id 6-13, 6-14
activating
objects 6-12, 6-15
POA Manager 6-18, 6-20
activation
implicit 6-16
on demand 6-15
Active Object Map 6-2
deactivate object 6-21
active state 6-18
POA Manager 6-19
adapter activators
using 6-10
AdapterActivator 6-10
AdapterAlreadyExists 6-4
add_ior_component method 7-6
add_ior_component_to_profile method 7-6
-all 3-4
application-assigned object IDs 6-14
arrays
data types
Java RMI
arrays 9-2

B

building an RMI-IIOP client program 9-4
invoke operations 9-5
locate object reference 9-4
building an RMI-IIOP server program 9-2
define interface 9-2
generate skeleton classes 9-3
implement interface 9-3
implement server 9-3

C

Calculator Program
client-server example 2-1
developing programs 2-5, 2-9
Calculator Client Program 2-10
Calculator Server Program 2-6
IDL 2-1
com.twoab.orb2.security.access.control.operations 4-8
com.twoab.orb2.security.audit.class 4-8
com.twoab.orb2.security.audit.file 4-8
com.twoab.orb2.security.audit.invocation 4-8
com.twoab.orb2.security.audit.principal.auth 4-8
com.twoab.orb2.security.audit.session.auth 4-8
com.twoab.orb2.security.audit.success 4-8
com.twoab.orb2.security.authenticate.client 4-8
com.twoab.orb2.security.authentication.class 4-8
com.twoab.orb2.security.center.host 4-8
com.twoab.orb2.security.center.port 4-8
com.twoab.orb2.security.cipher.suites 4-9
com.twoab.orb2.security.create.identity 4-9
com.twoab.orb2.security.debug 4-9
com.twoab.orb2.security.instance 4-9
com.twoab.orb2.security.key.algorithm 4-9
com.twoab.orb2.security.key.provider.class 4-9
com.twoab.orb2.security.key.store.password 4-9
com.twoab.orb2.security.key.store.path 4-9
com.twoab.orb2.security.key.store.provider.class 4-9
com.twoab.orb2.security.key.store.type 4-9
com.twoab.orb2.security.name 4-9
com.twoab.orb2.security.password 4-10
com.twoab.orb2.security.port 4-10
com.twoab.orb2.security.stateless 4-10
com.twoab.orb2.security.trust.algorithm 4-10
com.twoab.orb2.security.trust.identity 4-10
com.twoab.orb2.security.trust.provider.class 4-10
com.twoab.orb2.security.trust.store.password 4-10
com.twoab.orb2.security.trust.store.path 4-10
com.twoab.orb2.security.trust.store.provider.class 4-10
com.twoab.orb2.security.trust.store.type 4-11
com.twoab.orb2.server.request.interceptor.order 4-7
com.twoab.orb2.ServerRequestInterceptorOrder=7-2
com.twoab.orb2.Trader 10-9
com.twoab.orb2.trader.hostname 4-11
com.twoab.orb2.trader.port 4-11
com.twoab.orb2.trader2.hostname 4-11
com.twoab.orb2.trader2.port 4-11
com.twoab.orb2.Trading 10-10
com.twoab.orb2.util.IORFormat 10-1
com.twoab.orb2.util.IORFormatException 10-3
com.twoab.orb2.util.IORPrint 10-4
com.twoab.orb2.util.ProfileFormat 10-5
com.twoab.orb2.util.SystemExceptionFormat 10-6
com.twoab.orb2.util.TaggedProfileFormat 10-8
com.twoab.orb2.wchar_set 4-7, 9-1
compiler
  idlc 2-3, 3-1, 3-5
  running 3-1, 3-5
  running w/o idlc scripts 3-4, 3-6
Compiling IDL 2-2
cookie 6-10
CORBA Interoperability 1-1
CORBA object references 9-2
CORBA objects
  persistent 6-12
  transient 6-12
CORBA Object 2-6
CORBA::BAD_INV_ORDER 2-5
CORBA::Current 6-7
CORBA::OBJECT_NOT_EXIST 6-12, 6-15
CORBA::ORB 2-6
CORBA-IIOP client 9-5
CORBA-IIOP service 9-5
CPOA Interface Operations
  conversions 6-2
create_POA 6-4
Current interface 6-7
D
-D 4-1
-D a_symbol 3-2, 3-5
-d directory_name 3-1, 3-5
DAIS_SecondaryTrader 4-5
deactivate object
  Active Object Map 6-21
  deactivate operation
    POA Manager 6-19
deactivate_object 6-8, 6-21
  deactivate_object operation
    POA 6-21
deactivating objects 6-21
default servants 6-10
degregation 2-4, 2-9
  Delegation (Jie-based) Servants 6-7
degregation-based servers 2-4
demonstration programs 1-3
  Calculator Server Program 2-1
  Demos.CalculatorPOA 2-9
developing
  Calculator Client Program 2-10
  Calculator Server Program 2-6
  client programs 2-9
distributed applications, principals 2-1
  servants 2-5
  server programs 2-5
discard_requests operation
  POA Manager 6-19
discarding state 6-18
  POA Manager 6-19
distributed applications developing 2-1
DSI interface 6-10
default servants 6-10
Dynamic Any’s 8-1
Dynamic Interface 8-1
Dynamic Invocation Interface 8-1
Dynamic Requests 8-1
invoking 8-1
receiving requests 8-1
Dynamic Skeleton Interface 8-1

E

ending interception points 7-4
environment variables
    ORB2 1-1
    ORB2C 1-1
    PATH 1-1
establish_components method 7-6
etherealization of servants
definition 6-7
etherealize 6-8, 6-9, 6-21
etherealize_objects 6-21
examples
    registering request interceptors 7-2
exceptions 9-2
    CORBA::BAD_INV_ORDER 2-5
    forward request 6-15, 7-4
    system 7-4

F

files
    orb.properties
    ordering request interceptors 7-2
find_POA 6-10
FooPOA 6-14
forward request exception 6-15, 7-4

G

get_servant 6-10
get_state operation
    POA Manager 6-19

H

-help 3-1, 3-5, 3-6
Helper class 6-3
holding state 6-18
    POA Manager 6-19

I

-I directory_name 3-2, 3-5
ID Assignment policy 6-6
IDL
    compiling 2-2
    IDL Compiler
        idl 2-3, 3-1, 3-5
        options 3-1, 3-5
        running 3-1, 3-5
        running w/o idlc scripts 3-4, 3-6
    IDL entity types 9-2
    idlc compiler 2-3, 3-1, 3-5
    running 3-1, 3-5
    running w/o idlc scripts 3-4, 3-6
    implementing operations 2-5
implicit activate objects 6-16
IMPLICIT_ACTIVATION 6-4, 6-6, 6-13, 6-16
    policy 6-6, 6-16
inactive state 6-18
    POA Manager 6-19
incarnate 6-8, 6-9
incarnation of servants
definition 6-7
    -include_pdk 3-3
    -include_pdk_file 3-3
increment method 7-9
inheritance 2-4, 10-10
inheritance-based
    servants 6-7
    servers 2-4
init 2-6, 2-10, 6-4
Initializer 7-6
Initializing the ORB 4-1
    command-line arguments 4-2
    precedence of properties 4-2
    property files 4-1
    writing initialization software 4-1
interception points 7-3
    client 7-3
    ending 7-4
    intermediate 7-4
    server 7-3
    starting 7-4
Interceptor::name() method 7-3
interceptors
    IOR
        definition 7-1
        registering 7-5
        naming 7-3
        ordering 7-2
        portable 7-1
        definition 7-1
        registering example 7-2
        request 7-1
        definition 7-1
        registering 7-2
InterceptorsExampleServant inner class 7-9
Interface Definition Language. See IDL
interface POA 6-1
interface-namePOA 2-9
interfaces
    ClientRequestInterceptor 7-7
    Current 6-7
    DSI
default servants 6-10
IDL
    Calculator Server Program
    examples 2-1
IORInfo 7-5
IORInterceptor 7-5
ORBInitializer 7-1
Portable Object Adapter 2-4
PortableInterceptor::ClientRequestInterceptor
7-2
PortableInterceptor::ServerRequestInterceptor
7-2
servant 7-9
Servant Activator 6-8
ServantLocator 6-9
ServerRequestInterceptor 7-7
TraderOperations 10-3, 10-5, 10-6, 10-7
intermediate interception points 7-4
interoperability 1-1
into org.omg.CORBA.ORB.init 6-4
IOR interceptor
definition 7-1
invoking 7-5
registering 7-5
IOR interceptor points 7-5
IORFormat 10-1
IORInfo interface 7-5
IORInterceptor interface 7-5
IORInterceptor.establish_components method 7-5

J
Java
client application 2-10
Servant class 2-5
servant class 2-5
Java abstract class
Servant class 2-5
Java class
Java classpath 1-2
Java RMI data types 9-2
Java system properties
org.omg.CORBA.ORBClass 1-2
org.omg.CORBA.ORBSingletonClass 1-2

K
kill method 7-8, 7-9
KillServer class 7-8

L
Licensing 1-1
Lifespan policy 6-5, 6-12
lifetimes of objects and servants 6-12
locating objects
  Naming Service 1-2, 1-3
  Trader Service 1-2, 1-3
lookup_all 10-9
lookup_proxy_all 10-9
lookup_random 10-9

M
methods
  _this 6-16
  add_ior_component 7-6
  add_ior_component_to_profile 7-6
  callback 7-1
  establish_components 7-6
  increment 7-9
  Interceptor::name() 7-3
  interceptors 7-1
  IORInterceptor.establish_components 7-5
  kill 7-8, 7-9
  ORBInitializer::post_init 7-5
  ORBInitializer::pre_init 7-5
  ORBInitInfo::add_client_request_interceptor 7-2
  ORBInitInfo::add_iar_interceptor 7-5
  ORBInitInfo::add_server_request_interceptor 7-2
  post_init 7-1, 7-2, 7-5, 7-6
  postinvoke 6-9
  pre_init 7-1, 7-2, 7-5
  preinvoke 6-9
  resolve_initial_references 10-10
  string_to_object 1-2
  System.setProperty 1-2
MULTIPLE_ID 6-6
MULTIPLE_ID policy 6-6, 6-16
my_little_poa 6-5, 6-11

N
name interceptors 7-3
Naming Service 1-2, 1-3
NO_IMPLICIT_ACTIVATION 6-6
NO_IMPLICIT_ACTIVATION policy 6-6
NoContext 6-7
NON_RETAIN 6-6, 6-8, 6-10
NON-RETAIN policy 6-6
-noprefix 3-2
-noreverseprefix 3-2
-noskeleton 3-4
-nostub 3-4
-notie 3-4

O
OBJ_ADAPTER 6-19
Object Adapter 6-1
object IDs 6-12
  application-assigned 6-14
object references 9-2
object references, stringified 1-2
OBJECT_NOT_EXIST 6-4
object_to_string 1-2
ObjectID Uniqueness policy 6-6
objects
  activate persistent 6-17
  activating 6-12
  and servant association 6-8
  deactivating 6-21
  identifying 6-12
  IDs
    application-assigned 6-14
    POA-assigned 6-13
implementing multiple 6-7
implicit activation 6-16
implicitly activate 6-16
lifetime of 6-12
on demand activation 6-15
ORBInitializer 7-1, 7-2, 7-5
specifying 7-1
persistent 6-12
servant 6-6
Trading
obtaining 10-10
transient 6-12
OMG Specifications
Portable Interceptors Specification 7-2
on demand 6-15
OpenOrb 3-4
operations
activate
POA Manager 6-19
deactivate
POA Manager 6-19
deactivate_object
POA 6-21
discard_requests
POA Manager 6-19
get_state
POA Manager 6-19
implementation of 2-5
POA 6-2
POA::servant_to_id 6-16
POA::servant_to_reference 6-16
ORB
and POA interactions 6-1
associating servers with 2-5
ORB initialization
precedence of properties 4-2
property files 4-1
writing the software 4-1
ORB Properties 4-5
ORB.init 4-1, 4-5
orb.properties file
ordering request interceptors 7-2
ORB.run 6-21
ORB.shutdown 2-8
ORB::init 5-1
ORB_CTL_MODEL policy 6-5
ORB_CTRL_MODEL 6-4, 6-5, 6-6
ORB2 1-1
orb2 Security
options
-ORBsec 4-4
-ORBsec_debug 4-4
-ORBsec_key_path 4-4
-ORBsec_key_pw 4-4
-ORBsec_port 4-4
-ORBsec_trust_path 4-4
-ORBsec_trust_pw 4-4
properties
com.twoab.orb2.security 4-7
com.twoab.orb2.security.access.control.operations 4-8
com.twoab.orb2.security.audit.class 4-8
com.twoab.orb2.security.audit.file 4-8
com.twoab.orb2.security.audit.invocation 4-8
com.twoab.orb2.security.audit.principal.auth 4-8
com.twoab.orb2.security.audit.session.auth 4-8
com.twoab.orb2.security.audit.success 4-8
com.twoab.orb2.security.authenticate.client 4-8
com.twoab.orb2.security.authentication.class 4-8
com.twoab.orb2.security.center.host 4-8
com.twoab.orb2.security.center.port 4-8
com.twoab.orb2.security.cipher.suites 4-9
com.twoab.orb2.security.create.identity 4-9
com.twoab.orb2.security.debug 4-9
com.twoab.orb2.security.instance 4-9
com.twoab.orb2.security.key.algorithm 4-9
com.twoab.orb2.security.key.provider.class 4-9
com.twoab.orb2.security.key.store.password 4-9
com.twoab.orb2.security.key.store.path 4-9
com.twoab.orb2.security.key.store.provider.class 4-9
com.twoab.orb2.security.key.store.type 4-9
com.twoab.orb2.security.name 4-9
com.twoab.orb2.security.password 4-10
com.twoab.orb2.security.port 4-10
com.twoab.orb2.security.stateless 4-10
com.twoab.orb2.security.trust.algorithm 4-10
com.twoab.orb2.security.trust.identity 4-10
com.twoab.orb2.security.trust.provider.class 4-10
com.twoab.orb2.security.trust.store.password 4-10
com.twoab.orb2.security.trust.store.path 4-10
com.twoab.orb2.security.trust.store.provider.class 4-10
com.twoab.orb2.security.trust.store.type 4-11
com.twoab.orb2.security.user 4-11
orb2.jar 1-2
ORB2C 1-1
-ORBbidir_iiop 4-2
-ORBclient_request_interceptor_order 4-2
-ORBconfig 4-2
-ORBdebug 4-2
-ORBdebug_file 4-2
ORBgiop_minor_version 4-3
-ORBhost 4-3
-ORBid 4-3
ORBInitializer 7-6
ORBInitializer class 7-1
ORBInitializer interface 7-1
ORBInitializer object 7-1, 7-2
specifying 7-1
ORBInitializer objects 7-5
ORBInitializer::post_init method 7-5
ORBInitializer::pre_init method 7-5
ORBInitInfo class 7-2, 7-5
ORBInitInfo::add_client_request_interceptor method 7-2
ORBInitInfo::add_ior_interceptor method 7-5
ORBInitInfo::add_server_request_interceptor method 7-2
-ORBInitRef 4-3
-ORBlicense_dir 4-3
-ORBmaxthreads 4-3
-ORBminthreads 4-3
-ORBport 4-3, 6-17
-ORBrequest_timeout 4-3
-ORBsec 4-4
-ORBsec_debug 4-4
-ORBsec_key_path 4-4
-ORBsec_key_pw 4-4
-ORBsec_port 4-4
-ORBsec_trust_path 4-4
-ORBsec_trust_pw 4-4
-ORBserver_request_interceptor_order 4-3
-ORBthreadtimeout 4-4
ordering request interceptors 7-2
org.omg.CORBA.Object 6-3
org.omg.CORBA.ORB 2-10
org.omg.CORBA.ORB.init 6-6
org.omg.CORBA.ORBClass 1-2
org.omg.CORBA.ORBSingletonClass 1-2
org.omg.CORBA_2_3.ORB.set_delegate 2-6
org.omg.PortableServer.POA 2-6
activating 6-18, 6-20
states 6-18, 6-19
Manager states 6-18
object identification within a 6-12
operations 6-2
policies 6-5
PortableServer module 2-5
registering a servant 6-6
Root 6-2, 6-3
ServantActivator::erealize 6-21
using a custom 6-4
using adapter activators 6-10
POA Interface Operations 6-2
Attributes 6-2
Default Servant Operations 6-2
Object Activation/ Deactivation 6-2
Object Reference Creation 6-2
POA Lifecycle Management 6-2
Policy Creation 6-2
Servant Manager Operations 6-2
POA Manager
activating 6-20
states 6-19
POA::activate_object 2-5
POA::activate_object_with_id 2-5
POA::create_POA 6-4
POA::servant_to_id 6-16
POA::servant_to_reference 6-16
POAHelper.narrow 2-6
POAManager 2-7, 6-5
POAManager::activate 6-18
policies
MULTIPLE_ID 6-16
PERSISTENT 6-17
POA 6-5
UNIQUE_ID 6-16
USER_ID 6-17
portable interceptors 7-1
IOR interceptors 7-1
request interceptors 7-1
Portable Object Adapter (POA) interface 2-4
Portable Object Adapter. See POA
PortableInterceptor::ClientRequestInterceptor interface 7-2
PortableInterceptor::ServerRequestInterceptor interface 7-2
PortableServer module 2-5
PortableServer::Current 6-7
PortableServer::ServantActivator::incarnate() 2-6
PortableServer::ServantLocator 6-9
PortableServer::ServantLocator::preinvoke() 2-6
post_init method 7-1, 7-2, 7-5, 7-6
postinvoke 6-9
postinvoke method 6-9
pre_init method 7-1, 7-2, 7-5
preinvoke 6-9
preinvoke method 6-9
primitive types
data types
Java RMI
primitive types 9-2
println 7-7
processing states of POA Manager 6-18
ProfileFormat 10-5
programming, APIs 5-2
proxy object 2-2

R
receive_exception 7-3
receive_other 7-3
receive_reply 7-3
receive_request 7-3, 7-4
receive_request_service_context 7-3, 7-4
reference
obtaining an object reference 6-12
register
IOR interceptor 7-5
request interceptor 7-1, 7-2
-release 3-1, 3-5, 3-6
Remote Method Invocation. See RMI-IIOP
request interceptor
definition 7-1
invoking 7-4
registering 7-1, 7-2
Request Processing policy 6-6
requests
dynamic 8-1
invoking 8-1
receiving 8-1
static requests 8-1
resolve_initial_references 2-6, 4-4, 4-5, 6-3, 10-9, 10-10
RETAIN 6-4, 6-6, 6-8, 6-13, 6-15, 6-16, 6-21
RETAIN policy 6-6
RMI codebase 9-5
RMI remote interfaces 9-2
RMI-IIOP 9-1
building a client program 9-4
invoke operations 9-5
locate object reference 9-4
building a server program 9-2
define interface 9-2
generate skeleton classes 9-3
implement interface 9-3
implement server 9-3
CORBA-IIOP client 9-5
CORBA-IIOP service 9-5
RMI codebase 9-5
RMI-IIOP client 9-5
RMI-IIOP server 9-5
using with orb2 for Java 9-1
RMI-IIOP client 9-5
RMI-IIOP server 9-5
RootPOA 6-2, 6-3, 7-8
using the 6-3

S
SECONDARY_TRADER_PORT 4-5
send_exception 7-3
send_other 7-3
send_poll 7-3, 7-4, 7-7
send_reply 7-3, 7-4
send_request 7-3, 7-4
Servant Activator interface 6-8
servant class 2-5
servant interface 7-9
servant managers, using 6-8
servant retention policy 6-6
ServantActivator 6-8, 6-9
ServantActivator::etherealize
POA 6-21
ServantLocator 6-9, 6-10
ServantLocator interface 6-9
ServantManager 2-6, 6-8
ServantRetention 6-9
servers 6-1
acquiring 6-6
associating with an ORB instance 2-5
default
DSI interface 6-10
devolution-based 2-9, 6-7
devolution-based servants 6-7
developing 2-5
caractizing and etherealizing 6-7
inheritance-based 6-7
inheriting-based servants 6-7
postivoking 6-9
preinvoking 6-9
using default 6-10
server
interception points 7-3
killing 7-8
Server class 7-8
server interception points
receive_request 7-3
receive_request_service_context 7-3
send_exception 7-3
send_other 7-3
send_reply 7-3
server programs, developing 2-5
ServerInterceptor class 7-7
ServerRequestInterceptor interface 7-7
servers
associating with an ORB 2-5
development-based 2-4
heritance-based 2-4
ServerShutdown 7-8
set_servant 6-10
set_servant_manager 6-13
shutdown 6-8
SINGLE_THREAD_MODEL 6-5
SINGLE_THREAD_MODEL policy 6-5
skeleton code 2-2
starting interception points 7-4
state 6-10
states
  active 6-18
    POA Manager 6-19
discarding 6-18
    POA Manager 6-19
holding 6-18
    POA Manager 6-19
inactive 6-18
    POA Manager 6-19
POA Manager 6-19
Static Requests 8-1
string_to_object 1-2
Stringified Object References 1-2
stub code 2-2
  proxy objects 2-2
system exception 7-4
system properties, Java
  org.omg.CORBA.ORBClass 1-2
  org.omg.CORBA.ORBSingletonClass 1-2
System.setProperty 1-2
SYSTEM_ID 6-4, 6-6, 6-13, 6-16
SYSTEM_ID policy 6-6, 6-13
SystemExceptionFormat 10-6

T
TaggedComponentFormat 10-7
TaggedProfileFormat 10-8
Thread policy 6-5
Tie class 2-9
TLS 4-4
TLS_UP 4-4
TLS_UP_CLIENT_ONLY 4-4, 4-7
Trace 7-6
Trader 10-3, 10-6, 10-7
Trader Service 1-2, 1-3
  command-line arguments 4-4
    SECONDARY_TRADER_PORT 4-5
    TRADER_PORT 4-5
options
  TRADER_ADDRESS 4-4, 4-5
properties 4-11
  com.twoab.orb2.trader.hostname 4-11
  com.twoab.orb2.trader.port 4-11
  com.twoab.orb2.trader2.hostname 4-11
  com.twoab.orb2.trader2.port 4-11
TRADER_ADDRESS 4-4, 4-5
TRADER_PORT 4-5
TraderOperations class 10-9
TraderOperations interface 10-3, 10-5, 10-6, 10-7
Trading class 10-10
Trading object
  obtaining 10-10
TRANSIENT 6-4, 6-5, 6-12, 6-19
transient
  CORBA object 6-12
TRANSIENT policy 6-5

U
UNIQUE_ID 6-4, 6-6
UNIQUE_ID policy 6-6, 6-16
unknown_adapter 6-11
UP_CLIENT_ONLY 4-4
USE_ACTIVE_OBJECT_MAP_ONLY 6-4, 6-6
USE_ACTIVE_OBJECT_MAP 6-7
USE_ACTIVE_OBJECT_MAP_ONLY policy 6-6
USE_DEFAULT_SERVANT 6-6, 6-7, 6-10
USE_DEFAULT_SERVANT policy 6-6, 6-10
USE_SERVANT_MANAGER 6-6, 6-7, 6-8, 6-13, 6-15
USE_SERVANT_MANAGER policy 6-6
USE_SERVANT_MANGER 6-8
USER_ID 6-5, 6-6, 6-8, 6-14
USER_ID policy 6-6, 6-14, 6-17
using RMI-IOP with orb2 for Java 9-1

V

value types
data types
  Java RMI
  value types 9-2

W

wait_for_completion 6-21