THE ENTERPRISE MIDDLEWARE SOLUTION

BEA WebLogic Enterprise

Java Programming Reference

BEA WebLogic Enterprise 4.2
Document Edition 4.2
July 1999

Java Programming Reference

Document Edition	Date	Software Version
4.2	July 1999	BEA WebLogic Enterprise 4.2

Contents

6. Security Service

Preface

Purpose of This Document

This document provides Java programmer reference information for the following BEA WebLogic Enterprise (sometimes referred to as WLE) product components:

- OMG IDL
- Server Description File
- TP Framework
- Bootstrap object
- FactoryFinder
- Security Service
- Transaction Service
- Interface Repository
- Application build and administration commands
- CORBA ORB
- IDL to Java mapping

The information provided in this document is supplemented by the *Java API Reference*, which contains descriptions of the application programming interface (API) for the following components:

- TP Framework
- Bootstrap object
- FactoryFinder
- Security Service
- Java Transaction Service (JTS)
- Java Transaction API (JTA)

Note: Effective February 1999, the BEA M3 product is renamed. The new name of the product is BEA WebLogic Enterprise (WLE).

Who Should Read This Document

This document is intended for application developers interested in using the WebLogic Enterprise software to write the following applications:

- Server applications implemented in the Java programming language
- All client applications supported by the WebLogic Enterprise product

This document assumes a familiarity with CORBA and Java programming. For reference information about implementing WebLogic Enterprise server applications in the C++ programming language, see the *C++ Programming Reference*.

How This Document Is Organized

The *Java Programming Reference* is organized as follows:

- Chapter 1, “OMG IDL Syntax,” provides a brief discussion on the Object Management Group (OMG) Interface Definition Language (IDL), and includes a cross-reference to a recommended publication about OMG IDL coding style guidelines.

- Chapter 2, “Server Description File,” describes the Server Description File.
- Chapter 3, “TP Framework,” includes high-level programming topics relevant to the WebLogic Enterprise TP Framework.
- Chapter 4, “Bootstrap Object,” describes the Bootstrap object.
- Chapter 5, “FactoryFinder Interface,” describes the FactoryFinder interface.
- Chapter 6, “Security Service,” describes the Security Service.
- Chapter 7, “Transaction Service,” describes the Transaction Service.
- Chapter 8, “Interface Repository Interfaces,” describes the Interface Repository interfaces.
- Chapter 9, “Joint Client/Server Applications,” describes programming requirements for joint client/servers.
- Chapter 10, “Java Development and Administration Commands,” describes the development and administration commands for WebLogic Enterprise applications on UNIX and Windows NT platforms.
- Chapter 11, “CORBA ORB,” provides a number of programming topics related to using the CORBA ORB. The information provided in this chapter is supplementary to the Sun Microsystems, Inc. documentation of the `org.omg.CORBA` package API, which is available in the Java Development Kit (JDK) 1.2.
- Chapter 12, “Mapping IDL to Java,” contains reprints on select topics on mapping IDL to Java from the Java IDL documentation published by Sun Microsystems, Inc.

How to Use This Document

This document, *Java Programming Reference*, is designed primarily as an online, hypertext document. If you are reading this as a paper publication, note that to get full use from this document you should access it as an online document via the Online Documentation CD for the BEA WebLogic Enterprise 4.2 release.

The following sections explain how to view this document online, and how to print a copy of this document.

Opening the Document in a Web Browser

To access the online version of this document, open the following file:

```
\doc\wle\v42\index.htm
```

Note: The online documentation requires Netscape Communicator version 4.0 or later, or Microsoft Internet Explorer version 4.0 or later.

Printing from a Web Browser

You can print a copy of this document, one file at a time, from the Web browser. Before you print, make sure that the chapter or appendix you want is displayed and *selected* in your browser. To select a chapter or appendix, click anywhere inside the chapter or appendix you want to print.

The Online Documentation CD includes Adobe Acrobat PDF files of all of the online documents. You can use the Adobe Acrobat Reader to print all or a portion of each document. On the CD Home Page, click the PDF Files button and scroll to the entry for the document you want to print.

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Item
boldface text	Indicates terms defined in the glossary.
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.
italics	Indicates emphasis or book titles.

Convention	Item

`monospace text`	Indicates code samples, commands and their options, data structures and their members, data types, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`.doc`
	`BITMAP`
	`float`

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Examples*: LPT1 SIGNON OR
`{ }`	Indicates a set of choices in a syntax line. The braces themselves should never be typed.

`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed.
	Example:
	`buildobjclient [-v] [-o name ] [-f` *`file-list`*`]...`
	`[-l` *`file-list`*`]...`

`|`	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.

Convention	Item
`...`	Indicates one of the following in a command line: ♦ That an argument can be repeated several times in a command line ♦ That the statement omits additional optional arguments ♦ That you can enter additional parameters, values, or other information The ellipsis itself should never be typed. *Example*: `buildobjclient [-v] [-o name ] [-f file-list]... [-l file-list]...`
. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

Related Documentation

The following sections list the documentation provided with the BEA WebLogic Enterprise software, related BEA publications, and other publications related to the technology.

BEA WebLogic Enterprise Documentation

The BEA WebLogic Enterprise information set consists of the following documents:

Installation Guide

C++ Release Notes

Java Release Notes

Getting Started

Guide to the University Sample Applications

Guide to the Java Sample Applications

Creating Client Applications

Creating C++ Server Applications

Creating Java Server Applications

Administration Guide

Using Server-to-Server Communication

C++ Programming Reference

Java Programming Reference (this document)

Java API Reference

JDBC Driver Programming Reference

System Messages

Glossary

Technical Articles

Note: The Online Documentation CD also includes Adobe Acrobat PDF files of all of the online documents. You can use the Adobe Acrobat Reader to print all or a portion of each document.

BEA Publications

Selected BEA TUXEDO Release 6.5 for BEA WebLogic Enterprise version 4.2 documents are available on the Online Documentation CD.

To access these documents:

1. Click the Other Reference button from the main menu.
2. Click the TUXEDO Documents option.

Other Publications

For more information about CORBA, Java, and related technologies, refer to the following books and specifications:

Cobb, E. 1997. *The Impact of Object Technology on Commercial Transaction Processing*. VLDB Journal, Volume 6. 173-190.

Edwards, J. with DeVoe, D. 1997. *3-Tier Client/Server At Work*. Wiley Computer Publishing.

Edwards, J., Harkey, D., and Orfali, R. 1996. *The Essential Client/Server Survival Guide*. Wiley Computer Publishing.

Flanagan, David. May 1997. *Java in a Nutshell*, 2nd Edition. O'Reilly & Associates, Incorporated.

Flanagan, David. September 1997. *Java Examples in a Nutshell*. O'Reilly & Associates, Incorporated.

Fowler, M. with Scott, K. 1997. *UML Distilled, Applying the Standard Object Modeling Language*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series.

Jacobson, I. 1994. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley.

Mowbray, Thomas J. and Malveau, Raphael C. (Contributor). 1997. *CORBA Design Patterns*, Paper Back and CD-ROM Edition. John Wiley & Sons, Inc.

Orfali, R., Harkey, D., and Edwards, J. 1997. *Instant Corba*. Wiley Computer Publishing.

Orfali, R., Harkey, D. February 1998. *Client/Server Programming with Java and CORBA*, 2nd Edition. John Wiley & Sons, Inc.

Otte, R., Patrick, P., and Roy, M. 1996. *Understanding CORBA*. Prentice Hall PTR.

Rosen, M. and Curtis, D. 1998. *Integrating CORBA and COM Applications*. Wiley Computer Publishing.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Loresen, W. 1991. *Object-Oriented Modeling and Design*. Prentice Hall.

The Common Object Request Broker: Architecture and Specification. Revision 2.2, February 1998. Published by the Object Management Group (OMG).

CORBAservices: Common Object Services Specification. Revised Edition. Updated: November 1997. Published by the Object Management Group (OMG).

Contact Information

The following sections provide information about how to obtain support for the documentation and the software.

Documentation Support

If you have questions or comments on the documentation, you can contact the BEA Information Engineering Group by e-mail at **docsupport@beasys.com**. (For information about how to contact Customer Support, refer to the following section.)

Customer Support

If you have any questions about this version of the BEA WebLogic Enterprise product, or if you have problems installing and running the BEA WebLogic Enterprise software, contact BEA Customer Support through BEA WebSupport at `www.beasys.com`. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address

- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

CHAPTER

1 OMG IDL Syntax

The Object Management Group (OMG) Interface Definition Language (IDL) is used to describe the interfaces that client objects call and that object implementations provide. An OMG IDL interface definition fully specifies each operation's parameters and provides the information needed to develop client applications that use the interface's operations.

Client applications are written in languages for which mappings from OMG IDL statements have been defined. How an OMG IDL statement is mapped to a client language construct depends on the facilities available in the client language. For example, an OMG IDL exception might be mapped to a structure in a language that has no notion of exception, or to an exception in a language that does.

OMG IDL statements obey the same lexical rules as C++ statements, although new keywords are introduced to support distribution concepts. OMG IDL statements also provide full support for standard C++ preprocessing features and OMG IDL-specific pragmas.

The OMG IDL grammar is a subset of ANSI C++ with additional constructs to support the operation invocation mechanism. OMG IDL is a declarative language; it supports C++ syntax for constant, type, and operation declarations; it does not include any algorithmic structures or variables.

For a description of OMG IDL grammar, see Chapter 3 of the *Common Object Request Broker: Architecture and Specification* Revision 2.2 "OMG IDL Syntax and Semantics."

Style Guidelines for Writing OMG IDL Statements

Refer to the following publication for OMG IDL style guidelines:

Mowbray, Thomas J. and Malveau, Raphael C.(Contributor). 1997.
CORBA Design Patterns, Paper Back and CD-ROM Edition.
John Wiley & Sons, Inc.

OMG IDL Extensions

The IDL compiler defines preprocessor macros specific to the platform. All the macros predefined by the preprocessor that you are using can be used in the OMG IDL file, in addition to the user-defined macros. You can also define your own macros when you are compiling or loading OMG IDL files.

CHAPTER

2 Server Description File

This chapter contains the following topics:

- Creating the Server Description File. This section includes the following topics:
  - About Object Activation and Deactivation
  - Server Description File Syntax
- Sample Server Description File

When you create a Java server application meant to be run in the WebLogic Enterprise environment, the `buildjavaserver` command accepts the following information:

- Default activation and transaction policies for all the objects implemented in the server application
- The server declaration, which includes the name of the Server object and the name of the server descriptor file
- The declarations of each of the modules and interfaces defined in the server application's OMG IDL file
- Nondefault activation and transaction policies for specific objects implemented in the server application
- A description of the content of the server application's `jar` archive, which contains all the files needed by the server application

You specify all the preceding information in a Server Description File, which is used by the `buildjavaserver` command to create the server descriptor file and, optionally, build a server `jar` file.

Creating the Server Description File

The means to provide the information required by the `buildjavaserver` command is the Server Description File, which is expressed in the XML language. XML looks very similar to HTML; its key difference is that no XML tag is predefined. Every XML file uses a Document Type Definition (DTD) file that specifies:

- ♦ What the XML tags are
- ♦ What attributes can be attached to an element
- ♦ What elements can be used in other elements

The DTD required by the WebLogic Enterprise system is packaged with the WebLogic Enterprise software. You create the Server Description File using a common text editor. The section “About Object Activation and Deactivation” on page 2-2 provides important background information about the policies you define in the Server Description File, and the section “Server Description File Syntax” on page 2-3 provides the details on how to specify the server description information in a Server Description File.

About Object Activation and Deactivation

The WebLogic Enterprise TP Framework application programming interface (API) provides callback methods for object activation and deactivation. These methods provide the ability for application code to implement flexible state management schemes for CORBA objects.

State management is the way you control the saving and restoring of object state during object deactivation and activation. State management also affects the duration of object activation, which influences the performance of servers and their resource usage. The external API of the TP Framework includes the `com.beasys.Tobj_Servant.activate_object` and `com.beasys.Tobj_Servant.deactivate_object` methods, which provide a possible location for state management code. Additionally, the TP Framework API includes the `com.beasys.Tobj.TP.deactivateEnable` method to enable the user to control the timing of object deactivation. The default duration of object activation is controlled by policies assigned to implementations when the server application is built by the `buildjavaserver` command.

While CORBA objects are active, their state is contained in a servant. This state must be initialized when objects are first invoked (that is, the first time a method is invoked on a CORBA object after its object reference is created) and on subsequent invocations after objects have been deactivated.

While a CORBA object is deactivated, its state must be saved outside the process in which the servant was active. When an object is activated, its state must be restored. The object's state can be saved in shared memory, in a file, in a database, and so forth. It is up to the programmer to determine what constitutes an object's state, and what must be saved before an object is deactivated, and restored when an object is activated.

You can use the Server Description File to set activation policies to control the duration of object activations in the server process. The activation policy determines the in-memory activation duration for a CORBA object. A CORBA object is active in a Portable Object Adapter (POA) if the POA's active object map contains an entry that associates an object ID with an existing servant. Object deactivation removes the association of an object ID with its active servant.

Server Description File Syntax

The Server Description File has the following four major parts:

- Prolog
- Server declaration
- Module and implementation declarations
- Archive declaration

The sections that follow explain the syntax and how to specify each of these parts of the Server Description File.

Prolog

Every Server Description File begins with the following required prolog:

```
<?xml version="1.0"?>
<!DOCTYPE M3-SERVER SYSTEM "m3.dtd">
```

If you want to override the default activation or transaction policy used by the `buildjavaserver` command, you can override those defaults in the prolog using the following syntax:

```
<?xml version="1.0"?>
<!DOCTYPE M3-SERVER SYSTEM "m3.dtd" [
       <!ENTITY TRANSACTION_POLICY "transaction_value">
       <!ENTITY ACTIVATION_POLICY "activation_value">
]>
```

In the preceding syntax, note the following:

- *`transaction_value`* represents one of the following: `never`, `ignore`, `optional`, or `always`. (Note that the double quotes are a required part of the syntax.)
- *`activation_policy`* represents one of the following: `method`, `transaction`, or `process`.
- The square brackets ([and]) preceding and following the `!ENTITY` tags are required; that is, the brackets in the preceding syntax do not imply that the enclosed text is optional.

Note that you specify default activation and transaction policies in the prolog only if you want to override the following WebLogic Enterprise system defaults:

Activation Policy	`method`
Transaction Policy	`optional`

Server Declaration

Immediately following the prolog is the server declaration, which is an optional part of the Server Description File. The server declaration contains the following:

- The fully qualified name of the Server object

- The fully qualified name of the file containing the server descriptor

To specify the server declaration, use the following syntax:

```
<M3-SERVER SERVER-IMPLEMENTATION="server_name"
           SERVER-DESCRIPTOR-NAME="server_descriptor">
</M3-SERVER>
```

In the preceding syntax, note the following:

- `server_name` represents the fully qualified name of the class that contains the Server object. Qualified names use dot separators, not slashes. If you do not specify the Server object, the WebLogic Enterprise system creates a default Server object that opens and closes the XA resource manager associated with the server application, if any, when the server application is started and stopped, respectively. (Note that the double quotes are a required part of the syntax.)

- `server_descriptor` represents the name of the file where the server descriptor will be stored. This file name typically has a `.ser` suffix. If you do not specify a server descriptor, the `buildjavaserver` command uses `Server.ser` by default.

Module and Implementation Declarations

After the prolog and the server declaration (if present), the Server Description File contains module and implementation declarations, which may be specified as nested elements.

The module declarations specify Java packages for the server application. Interface declarations specify:

- The interface repository ID for the interface being implemented

- Optionally, nondefault activation or transaction policies for objects that implement the interface

Module Declaration Syntax

A module declaration uses the following syntax:

```
<MODULE name="name">
     .
     .
     .
</MODULE>
```

In the preceding syntax, note the following:

- *name* represents the name of either a single Java package, or a set of nested packages. This variable is needed if it exists in the OMG IDL file, and it is used for scoping and grouping. Its use must be consistent with the way it is used inside the OMG IDL file.
- A module declaration can contain an implementation declaration, nested module declaration, or both.
- You can specify a nested package in a single module declaration using the dotted notation, or you can factor out the package name using nested module declarations. For example, either of the following module declarations for the `com.acme` package is valid:

  ```
  <MODULE name="com.acme">
       .
       .
       .
  </MODULE>
  ```

  or:

  ```
  <MODULE name="com">
       <MODULE name="acme">
             .
             .
             .
       </MODULE>
  </MODULE>
  ```

Implementation Declaration Syntax

An implementation declaration uses the following syntax:

```
<IMPLEMENTATION name="name"
                      [implements="interface_id"]
                      [transaction="transaction_policy"]
                      [activation="activation_policy"] />
```

In the preceding syntax, note the following:

- *name* represents the name of the implementation class. If the implementation declaration is not nested inside any module declaration, *name* must be the fully qualified class name, using the dotted notation.

If the implementation declaration is nested inside one or more module declarations, the names of the modules will be prepended to the implementation name to specify the whole name. The base class of the implementation name must be a skeleton class generated by the `m3idltojava` command.

- *`interface_id`* represents the IDL interface repository ID for the interface being implemented. This clause in the implementation declaration is optional. If you do not specify an interface ID, the WebLogic Enterprise system uses the most derived interface ID found in the skeleton class by default. The interface ID must match the most derived interface ID found in the skeleton class.
- *`transaction_policy`* represents the transaction policy used by the implementation in the server, and must be one of the keywords listed and described in the following table:

Policy	**Description**
`never`	The implementation is not transactional. Objects created for this interface can never be invoked within the scope of a transaction. The system generates an exception (`INVALID_TRANSACTION`) if an implementation with this policy is involved in a transaction. An `AUTOTRAN` policy specified in the `UBBCONFIG` file for the interface is ignored.
`ignore`	The implementation is not transactional. The system allows requests on this object to be made within the scope of a transaction, but the object is not part of the transaction. An `AUTOTRAN` policy specified in the `UBBCONFIG` file for the interface is ignored. (The BEA TUXEDO infrastructure always enforces the use of the `TPNOTRAN` flag (see `tpcall(3)` in the BEA TUXEDO System Reference) for requests associated with implementations that have this policy.
`optional`	The implementation may be transactional. Objects can be invoked either inside or outside the scope of a transaction. If the `AUTOTRAN` parameter is enabled in the `UBBCONFIG` file for the interface, the implementation is transactional. Servers containing transactional objects must be configured within a group associated with an XA-compliant RM.
`always`	The implementation is transactional. Objects are always transactional. If a request is made outside the scope of a transaction, the system automatically starts a transaction before invoking the method, and the transaction is committed when the method ends. (This is the `AUTOTRAN` feature.) Servers containing transactional objects must be configured within a group associated with an XA-compliant RM.

The transaction clause is optional. If you do not specify a transaction policy, the default is `optional`, unless the default value has been overridden in the prolog.

- ♦ *`activation_policy`* represents the activation policy used by the implementation in the server, and must be one of the keywords listed and described in the following table:

Policy	**Description**
`method`	The activation of the CORBA object (that is, the association between the object ID and the servant) lasts until the end of the method. At the completion of a method, the object is deactivated. When the next method is invoked on the object reference, the CORBA object is activated (the object ID is associated with a new servant). This behavior is similar to that of a BEA TUXEDO stateless service.
`transaction`	The activation of the CORBA object (that is, the association between the object ID and the servant) lasts until the end of the transaction. During the transaction, multiple object methods can be invoked. This is a model of resource allocation that is similar to that of a BEA TUXEDO conversational service. This model is less expensive than the BEA TUXEDO conversational service in that it uses fewer system resources. This is because of the WebLogic Enterprise ORB's multicontexted dispatching model (that is, the presence of many servants in memory at the same time for one server), which makes it possible for a single server process to be shared by many concurrently active servants, which service many clients. In the BEA TUXEDO system, the process would be dedicated to a single client and to only one service for the duration of a conversation.
`process`	The activation of the CORBA object (that is, the association between the object ID and the servant) lasts until the end of the process. **Note:** The TP Framework API provides an interface method (`com.beasys.Tobj.TP.deactivateEnable()`) that allows the application to control the timing of object deactivation for objects that have the `activation policy` set to `process`. For a description of this method, see the *Java API Reference*.

The activation policy determines the default in-memory activation duration for a CORBA object. A CORBA object is active in a POA if the POA's active object map contains an entry that associates an object ID with an existing servant. Object deactivation removes the association of an object ID with its active servant.

The activation clause is optional. If you do not specify an activation policy, the default is `method`, unless the default value has been overridden in the prolog.

Archive Declaration

The archive declaration describes the content of the `jar` archive that contains all the server application files. This section of the Server Description File is optional; if you do not provide this section, you can build the `jar` archive by using the `jar` command directly. However, declaring an archive in the Server Description File simplifies the process of collecting and identifying the files.

The archive declaration is the last section of the Server Description File. If you do not include an archive declaration, the `buildjavaserver` command produces only the server descriptor and places it in the file specified by the `server-descriptor-name` attribute in the server declaration.

You specify the content of the `<ARCHIVE>` element as either fully qualified Java classes or file names. When specifying file names, note that path specifications are system dependent, which has implications on archive portability.

The `buildjavaserver` command has the `searchpath` option, which you can use to specify the search path for the files and classes included in the archive.

Note: After you use the `buildjavaserver` command to create the `jar` archive, you might find it useful to verify the contents of the archive by using the `jar tvf` command. This helps make sure that the archive contains all the intended files.

Archive Declaration Syntax

The archive declaration has the following syntax:

```
<ARCHIVE name="archive-name">
      [<CLASS name="class-name" />] [...]
      [<PACKAGE name="package-name" />] [...]
      [<PACKAGE-RECURSIVE name="package-name"/>] [...]
      [<PACKAGE-ANONYMOUS />]
      [<FILE prefix="file-prefix" name="file-name" />] [...]
      [<DIRECTORY prefix="dir-prefix" name="dir-name" />] [...]
</ARCHIVE>
```

In the preceding syntax, note the following:

- Each of the entities nested inside the `<ARCHIVE>` element is optional, and there are no default values for any of these entities.
- The `[...]` construct next to an entity indicates that you can provide multiple such entities.
- *`archive-name`* represents the name of the `jar` archive file to be created by the `buildjavaserver` command. The archive created contains all the classes, packages, and files specified within the `<ARCHIVE>` element.
- *`class-name`* represents the fully qualified name of the class to be included in the archive. All inner classes of that class are included as well.
- *`package-name`* represents the fully qualified name of a package to be included in the archive. All the classes belonging to that package are included as well.

  If you want to include nested packages, use the `<PACKAGE-RECURSIVE>` element.
- Use the `<PACKAGE-ANONYMOUS>` element to specify that all classes not in a package are to be included in the archive. (This refers to the classes that do not have a `package` statement in the Java source.)
- *`file-name`* represents the name of a file to be included in the archive. You can use the *`file-prefix`* construct to specify a path name. This path name is prepended to the file name when the file is located to be included in the archive; however, the file is stored in the archive only with the name specified by *`file-name`*.

  For example, if the `file-name` is `acme/iconf.gif`, and the `file-prefix` is `/dev`, the `buildjavaserver` command looks for the file `/dev/acme/iconf.gif` and stores it in the archive as `acme/iconf.gif`.
- *`dir-name`* represents the path name of the directory to be included in the archive. All subdirectories are included as well. You can use the *`dir-prefix`* construct to specify a directory path. The directory path is prepended to the directory name when the directory is located to be included in the archive; however, the file is stored in the archive only with the name specified by *`dir-name`*.

Sample Server Description File

Listing 2-1 shows a sample Server Description File.

Listing 2-1 Sample Server Description File

```
<?xml version="1.0"?>
<!DOCTYPE M3-SERVER SYSTEM "m3.dtd"]>
<M3-SERVER
server-implementation="com.beasys.samples.BankAppServerImpl"
      server-descriptor-name="BankApp.ser">

       <MODULE name="com.beasys.samples">
              <IMPLEMENTATION
                     name="TellerFactoryImpl" />
                     activation="process"
                     transaction="never"
              />

              <IMPLEMENTATION
                     name="TellerImpl"/>
                     activation="method"
                     transaction="never"
              />

              <IMPLEMENTATION
                     name="DBAccessImpl"
                     activation="method"
                     transaction="never"
              />

       </MODULE>

       <ARCHIVE name="BankApp.jar">
              <PACKAGE name="com.beasys.samples"/>
       </ARCHIVE>
</M3-SERVER>
```

For an example of another Server Description File, see *Creating Java Server Applications*.

CHAPTER

3 TP Framework

This chapter contains the following topics:

- TP Framework Interfaces. This section describes the following interfaces:
  - Tobj_Servant Interface
  - Server Interface
  - TP Interface
- Transactions Usage Notes. This section describes the following topics:
  - Transaction Termination
  - Transaction Suspend and Resume
  - Restrictions
  - Voting on Transaction Outcome

The WebLogic Enterprise TP Framework provides a programming framework that enables users to create servers for high-performance TP applications. This chapter describes the architecture of and interfaces in the TP Framework. Information about the TP Framework API is in the *Java API Reference*. Information about how to use this API can be found in *Creating Java Server Applications*.

The TP Framework consists of:

- The `com.beasys.Tobj_Servant` class, which has virtual methods for object state management
- The `com.beasys.Tobj.Server` class, which has virtual methods for application-specific server initialization and termination logic
- The `com.beasys.Tobj.TP` class, which provides methods to:
  - Create object references for CORBA objects

  - Register (and unregister) factories with the FactoryFinder object
  - Initiate user-controlled deactivation of the CORBA object currently being invoked
  - Obtain an object reference to the CORBA object currently being invoked
  - Open and close XA resource managers
  - Log messages to a user log (`ULOG`) file
  - Obtain object references to the ORB and to Bootstrap objects
- Header files for these classes
- A library to be link-edited with server applications

TP Framework Interfaces

The TP Framework supports the following interfaces:

- `com.beasys.Tobj_Servant`
- `com.beasys.Tobj.Server`
- `com.beasys.Tobj.TP`
- `org.omg.CosTransactions.TransactionalObject` (deprecated in the previous release)

Tobj_Servant Interface

The `com.beasys.Tobj_Servant` interface defines operations that allow a CORBA object to assist in the management of its state. Every implementation skeleton generated by the IDL compiler automatically inherits from the `com.beasys.Tobj_Servant` class. The `com.beasys.Tobj_Servant` class contains two virtual methods, `activate_object` and `deactivate_object`, that can be redefined by the programmer.

Whenever a request comes in for an inactive CORBA object, the object is activated and the `activate_object` method is invoked on the servant. When the CORBA object is deactivated, the `deactivate_object` method is invoked on the servant. The timing of deactivation is driven by the implementation's activation policy. When `deactivate_object` is invoked, the TP Framework passes in a reason code to indicate why the call was made.

Note: The `activate_object` and `deactivate_object` methods are the only methods that the TP Framework guarantees will be invoked for CORBA object activation and deactivation. The servant class constructor and destructor may or may not be invoked at activation or deactivation time. Therefore, the server-application code must not do any state handling for CORBA objects in either the constructor or destructor of the servant class.

Server Interface

The `com.beasys.Tobj.Server` interface provides callback methods that can be used for application-specific server initialization and termination logic. The `com.beasys.Tobj.Server` class is a Java class.

Note: Unlike implementing C++ server applications with the WebLogic Enterprise system, when you are implementing Java server applications with the WebLogic Enterprise system, you must provide definitions for the `com.beasys.Tobj.Server.initialize` and `com.beasys.Tobj.Server.release` methods. The TP Framework provides default versions of these methods.

TP Interface

The `com.beasys.Tobj.TP` interface supplies a set of service methods that can be invoked by application code. This is the *only* interface in the TP Framework that can safely be invoked by application code. All other interfaces have callback methods that are intended to be invoked only by system code.

The purpose of this interface is to provide high-level calls that application code can call, instead of calls to underlying APIs provided by the Portable Object Adapter (POA), the CORBAservices Naming Service, and the BEA TUXEDO system. By using these calls, programmers can learn a simpler API and are spared the complexity of the underlying APIs.

The `com.beasys.Tobj.TP` interface implicitly uses two features of the WebLogic Enterprise software that extend the CORBA APIs:

- ♦ Factories and the FactoryFinder object
- ♦ Factory-based routing

Usage Notes

During server application initialization, the application constructs the object reference for an application factory. It then invokes the `register_factory` method, passing in the factory's object reference together with a factory `id` field. On server release (shutdown), the application uses the `unregister_factory` method to unregister the factory.

TransactionalObject Interface Not Enforced

The `org.omg.CosTransactions.TransactionalObject` interface was formerly used to indicate that an object was transactional. In the previous version of the WebLogic Enterprise software, if a transactional invocation was done on an object that did not descend from the `org.omg.CosTransactions.TransactionalObject` interface, an exception was raised. Therefore, in the previous version, an object had to descend from the `org.omg.CosTransactions.TransactionalObject` interface to be eligible to participate in transactions. This behavior was enforced by the TP Framework.

However, in version 2.1 of the WebLogic Enterprise software, this interface is deprecated. Therefore, the use of this interface is now optional and no enforcement of descent from this interface is done for objects infected with transactions. By specifying the `never` or `ignore` transaction policies, the programmer can specify that an object is not to be infected by transactions. There is no interface enforcement for eligibility for transactions. The only indicator is the transaction policy.

Note: The CORBAservices Object Transaction Service does not require that all requests be performed within the scope of a transaction. It is up to each object to determine its behavior when invoked outside the scope of a transaction; an object that requires a transaction context can raise a standard exception.

Transactions Usage Notes

The following sections provide some information about how to use transactions.

Transaction Termination

In general, the handling of the outcome of a transaction is the responsibility of the initiator. Therefore, the following is true:

♦ If the client or server application code initiates transactions, the TP Framework never commits a transaction. The WebLogic Enterprise system may roll back the transaction if server processing tries to return to the client with the transaction in an illegal state.

♦ If the system initiates a transaction, the commit or rollback will always be handled by the WebLogic Enterprise system.

The following behavior is enforced by the WebLogic Enterprise system:

♦ If no transaction is active when a method on a CORBA object is invoked and that method begins a transaction, the transaction must be either committed, rolled back, or suspended when the method invocation returns. If none of these actions is taken, the transaction is rolled back by the TP Framework and the `org.omg.CORBA.OBJ_ADAPTER` exception is raised to the client application.

  This exception is raised because the transaction was initiated in the server application; therefore, the client application would not expect a transactional error condition such as `TRANSACTION_ROLLEDBACK`.

Transaction Suspend and Resume

The CORBA object must follow strict rules with respect to suspending and resuming a transaction within a method invocation. These rules and the error conditions that result from their violation are described in this section.

When a CORBA object method begins execution, it can be in one of the following three states with respect to transactions:

- The client application began the transaction.
    - *Valid server application behavior:* Suspend and resume the transaction within the method execution.
    - *Invalid server application behavior:* Return from the method with the transaction in the suspended state (that is, return from the method without invoking resume if suspend was invoked).
    - *Error Processing:* If invalid behavior occurs, the TP Framework raises the `org.omg.CORBA.TRANSACTION_ROLLEDBACK` exception to the client application and the transaction is rolled back by the WebLogic Enterprise system.
- The system began a transaction to provide `AUTOTRAN` or transaction policy `always` behavior.

Note: For each CORBA interface, set `AUTOTRAN` to Yes if you want a transaction to start automatically when an operation invocation is received. Setting `AUTOTRAN` to Yes has no effect if the interface is already in transaction mode. For more information about `AUTOTRAN`, refer to the *Administration Guide*.

- *Valid server behavior:* Suspend and resume the transaction within the method execution.

  Note: Not recommended. The transaction may be timed out and aborted before another request causes the transaction to be resumed.

- *Invalid server behavior:* Return from the method with the transaction in the suspended state (that is, return from the method without invoking resume if suspend was invoked).
- *Error Processing:* If invalid behavior occurs, the TP Framework raises the `org.omg.CORBA.OBJ_ADAPTER` exception to the client and the transaction is rolled back by the system. The `org.omg.CORBA.OBJ_ADAPTER` exception is

raised because the client application did not initiate the transaction, and, therefore, does not expect transaction error conditions to be raised.

- The CORBA object is not involved in a transaction when it starts executing.
  - *Valid server behavior:*
    - Begin and commit a transaction within the method execution.
    - Begin and roll back a transaction within the method execution.
    - Begin and suspend a transaction within the method execution.
  - *Invalid server behavior:* Begin a transaction and return from the method with the transaction active.
  - *Error Processing:* If invalid behavior occurs, the TP Framework raises the `org.omg.CORBA.OBJ_ADAPTER` exception to the client application and the transaction is rolled back by the WebLogic Enterprise system. The `org.omg.CORBA.OBJ_ADAPTER` exception is raised because the client application did not initiate the transaction, and, therefore, does not expect transaction error conditions to be raised.

Restrictions

The following restrictions apply to WebLogic Enterprise transactions:

- A CORBA object in the WebLogic Enterprise system must have the same transaction context when it returns from a method invocation that it had when the method was invoked.
- A CORBA object can be infected by only one transaction at a time. If an invocation tries to infect an already infected object, an `org.omg.CORBA.INVALID_TRANSACTION` exception is returned.
- If a CORBA object is infected with a transaction and a nontransactional request is made on it, an `org.omg.CORBA.OBJ_ADAPTER` exception is raised.
- If the application begins a transaction in the `com.beasys.Tobj.Server.initialize` method, it must either commit or roll back the transaction before returning from the method. If it does not, the TP Framework shuts down the server. This is because the application has no predictable way of regaining control after completing the `initialize` method.

- If a CORBA object is infected by a transaction and with an activation policy of `transaction`, and if the reason code passed to the method is either `DR_TRANS_COMMITTING` or `DR_TRANS_ABORTED`, no invocation on any CORBA object can be done from within the `com.beasys.Tobj_Servant.deactivate_object` method. Such an invocation results in an `org.omg.CORBA.BAD_INV_ORDER` exception.
- If an object generates a user exception within a system-generated transaction (that is, the client did not begin a transaction explicitly), the client application receives the `org.omg.CORBA.OBJ_ADAPTER` system exception and not the user exception.

Voting on Transaction Outcome

CORBA objects can affect transaction outcome during two stages of transaction processing:

- *During transactional work*

  The `org.omg.CORBA.Current.rollback_only` method can be used to ensure that the only possible outcome is to roll back the current transaction. The `rollback_only` method can be invoked from any CORBA object method.

- *After completion of transactional work*

  CORBA objects that have the transaction activation policy are given a chance to vote whether the transaction should commit or roll back after transactional work is completed. These objects are notified of the completion of transactional work prior to the start of the two-phase commit algorithm when the TP Framework invokes its `deactivate_object` method.

  Note that this behavior does not apply to objects with process or method activation policies. If the CORBA object wants to roll back the transaction, it can invoke the `org.omg.CORBA.Current.rollback_only` method. If it wants to vote to commit the transaction, it does not make that call. Note, however, that a vote to commit does not guarantee that the transaction is committed, since other objects may subsequently vote to roll back the transaction.

Note: CORBA objects that have the `transaction` activation policy are, by definition, notified when a transaction completes, as described above. However, CORBA objects that have a `method` or `process` activation policy do not receive any notification. This is something that programmers need to be aware of.

For example, consider a CORBA object with activation policy set to `process` that opens an SQL cursor within a client-initiated transaction. Typically, once the client application commits the transaction, all cursors that were opened within that transaction are automatically closed; however, the object does not receive any notification that its cursor has been closed.

CHAPTER

4 Bootstrap Object

This chapter contains the following topics:

- How Bootstrap Objects Work
- Types of Remote Clients Supported
- Capabilities and Limitations
- Bootstrap Object API. This section describes the following:
  - Tobj Module
  - Java Mapping
- Programming Examples. The following examples are provided:
  - Getting a SecurityCurrent Object
  - Getting a UserTransaction Object

To communicate with WebLogic Enterprise objects, a client application must obtain object references. The client application uses the Bootstrap object to obtain initial object references to four key objects in a WebLogic Enterprise domain: the FactoryFinder (which is used to locate factory objects), SecurityCurrent (which is used to log on to the system), TransactionCurrent (which is used to manage transactions), and the Interface Repository (which is used to obtain information about available interfaces). However, this poses a problem: *How does the client application access the Bootstrap object?*

Bootstrap objects are local programming objects, not remote CORBA objects, in both the client and the server. When Bootstrap objects are created, their constructor requires the network address of a WebLogic Enterprise IIOP Server Listener/Handler. Given this information, the Bootstrap object can generate object references for the above-mentioned remote objects in the WebLogic Enterprise domain. These object references can then be used to access services available in the WebLogic Enterprise domain.

How Bootstrap Objects Work

Bootstrap objects are created by a client or a server application that must access object references to the following objects:

- SecurityCurrent
- TransactionCurrent
- FactoryFinder
- Interface Repository

In addition, you can use the Bootstrap object to return information needed by the client application; for example, information needed to initialize the client ORB.

Bootstrap objects represent the first connection to a specific WebLogic Enterprise domain. For a WebLogic Enterprise remote client, the Bootstrap object is created with the host and the port for the WebLogic Enterprise IIOP Server Listener/Handler. However, for WebLogic Enterprise native client and server applications, there is no need to specify a host and port because they execute in a specific WebLogic Enterprise domain. The IIOP Server Listener/Handler host and the port ID are included in the WebLogic Enterprise domain configuration information.

Once created, Bootstrap objects satisfy requests for object references for objects in a particular WebLogic Enterprise domain. Different Bootstrap objects allow the application to use multiple domains.

Using the Bootstrap object, you can obtain four different references, as follows:

- SecurityCurrent

  The SecurityCurrent object is used to establish a security context within a WebLogic Enterprise domain. The client can then obtain the PrincipalAuthenticator from the `principal_authenticator` attribute of the SecurityCurrent object.

- TransactionCurrent

  The TransactionCurrent object is used to participate in a WebLogic Enterprise transaction. The basic operations are as follows:

  - Begin

    Begin a transaction. Future operations take place within the scope of this transaction.

  - Commit

    End the transaction. All operations on this client application have completed successfully.

  - Roll back

    Abort the transaction. Tell all other participants to roll back.

  - Suspend

    Suspend participation in the current transaction. This operation returns an object that identifies the transaction and allows the client application to resume the transaction later.

  - Resume

    Resume participation in the specified transaction.

- FactoryFinder

  The FactoryFinder object is used to obtain a factory. In the WebLogic Enterprise system, factories are used to create application objects. The FactoryFinder provides the following different methods to find factories:

  - Get a list of all available factories that match a factory object reference (`find_factories`).
  - Get the factory that matches a name component consisting of id and kind (`find_one_factory`).
  - Get the first available factory of a specific kind (`find_one_factory_by_id`).
  - Get a list of all available factories of a specific kind (`find_factories_by_id`).
  - Get a list of all registered factories (`list_factories`).

- InterfaceRepository

  The Interface Repository contains the interface descriptions of the CORBA objects that are implemented within the WebLogic Enterprise domain. Clients using the Dynamic Invocation Interface (DII) need a reference to the Interface Repository to be able to build CORBA request structures. The ActiveX Client is a special case of this. Internally, the implementation of the COM/IIOP Bridge uses DII, so it must get the reference to the Interface Repository, although this is transparent to the desktop client.

The FactoryFinder and Interface Repository objects are not implemented in the environmental objects library. However, they are specific to a WebLogic Enterprise domain and are thus conceptually similar to the SecurityCurrent and TransactionCurrent objects in use.

You can also invoke the following methods on the Bootstrap object to return information needed by the client application:

- `getNativeProperties`

  This method returns the properties needed to initialize the ORB for native client applications. The `getNativeProperties` method must be invoked before any attempt is made to access any class in the `org.omg.CORBA` package.

- `getRemoteProperties`

  This method returns the properties needed to initialize the ORB for remote client applications.

- `getUserTransaction`

  This method returns the current transactional context object to the client application.

The Bootstrap object implies an association or "session" between the client application and the WebLogic Enterprise domain. Within the context of this association, the Bootstrap object imposes a containment relationship with the other Current objects (or contained objects); that is, the SecurityCurrent and TransactionCurrent. Current objects are valid only for this domain and only while the Bootstrap object exists.

In addition, a client can have only one instance of each of the Current objects at any time. If a Current object already exists, an attempt to create another Current object does not fail. Instead, another reference to the already existing object is handed out; that is, a client application may have more than one reference to the single instance of the Current object.

To create a new instance of a Current object, the application must first invoke the `destroy_current` method on the Bootstrap object. This invalidates all of the Current objects, but will not destroy the session with the WebLogic Enterprise domain. After invoking the `destroy_current` method, new instances of the Current objects can be created within the WebLogic Enterprise domain using the existing Bootstrap object.

To obtain Current objects for another domain, a different Bootstrap object must be constructed. Although it is possible to have multiple Bootstrap objects at one time, only one Bootstrap object may be "active," that is, have Current objects associated with it. Thus, an application must first invoke the `destroy_current` method on the "active" Bootstrap object before obtaining new Current objects on another Bootstrap object, which then becomes the active Bootstrap object.

Servers and native clients are inside of the WebLogic Enterprise domain; therefore, no "session" is established. However, the same containment relationships are enforced. Servers and native clients access the domain they are currently in by specifying an empty string, rather than by specifying `//host:port`. (When you compile client and server applications, specify the `-DTOBJADDR` option to specify a host and port to be used at run time, which allows for more flexibility and portability in client and server application code. For more information, see *Creating Client Applications* and *Creating Java Server Applications*.) Client and server applications must use the `com.beasys.Tobj_Bootstrap.resolve_initial_references` method, not the `org.omg.CORBA.ORB.resolve_initial_references` method.

Types of Remote Clients Supported

Table 4-1 shows the types of remote clients that can use the Bootstrap object to access the other environmental objects, such as FactoryFinder, SecurityCurrent, TransactionCurrent, and InterfaceRepository.

Table 4-1 Remote Clients Supported

Remote Client	Description
CORBA C++	CORBA C++ client applications use the WebLogic Enterprise C++ environmental objects to access the CORBA objects in a WebLogic Enterprise domain, and the WebLogic Enterprise Object Request Broker (ORB) to process from CORBA objects. Use the WebLogic Enterprise system development commands to build these client applications (see Chapter 10, "Java Development and Administration Commands").
CORBA Java	CORBA Java client applications use the Java environmental objects to access CORBA objects in a WebLogic Enterprise domain. However, these client applications use an ORB product other than the WebLogic Enterprise ORB to process requests from CORBA objects. These client applications are built using the ORB product's Java development tools. The Java core system of the WebLogic Enterprise software supports interoperability with client platforms using either of the following: ♦ The Java IDL ORB provided with the Java Development Kit 1.2 from Sun Microsystems, Inc. ♦ Netscape Communicator Version 4.0, using the bundled Visigenic IIOP-capable ORB For complete details about Java application and applet support, see the *Release Notes*.
ActiveX	Use the WebLogic Enterprise Automation environmental objects to access CORBA objects in a WebLogic Enterprise domain, and the ActiveX Client to process requests from CORBA objects. Use the Application Builder to create bindings for CORBA objects so that they can be accessed from ActiveX client applications. ActiveX client applications are built using a development tool such as Visual Basic, Delphi, or PowerBuilder.

This chapter describes how to use the Bootstrap object with Java client applications. For reference information about how to use the Bootstrap object in C++ and ActiveX client applications, see the *C++ Programming Reference*.

Capabilities and Limitations

Bootstrap objects have the following capabilities and limitations:

- Multiple Bootstrap objects can coexist in a client application, although only one Bootstrap object can own the Current objects (Transaction and Security) at one time. Client applications must invoke the `destroy_current` method on the Bootstrap object associated with one domain before obtaining the Current objects on another domain. Although it is possible to have multiple Bootstrap objects that establish connections to different WebLogic Enterprise domains, only one set of Current objects is valid. Attempts to obtain other Current objects without destroying the existing Current objects fail.
- Method invocations to any WebLogic Enterprise domain other than the domain that provides the valid SecurityCurrent object fail and return an `org.omg.CORBA.NO_PERMISSION` exception.
- Method invocations to any WebLogic Enterprise domain other than the domain that provides the valid TransactionCurrent object do not execute within the scope of a transaction.
- The transaction and security objects returned by the Bootstrap objects are BEA implementations of the Current objects. If other ("native") Current objects are present in the environment, they are ignored.

Bootstrap Object API

The Bootstrap object application programming interface (API) is described in the *Java API Reference*. The sections that follow describe:

- The object references returned by the Bootstrap object

- The Java mapping for the Bootstrap object
- Mappings for Microsoft desktop clients, including automation mapping

Tobj Module

Table 4-2 shows the object reference that is returned for each type ID.

Table 4-2 Returned Object References

ID	Returned Object Reference
FactoryFinder	FactoryFinder object (`com.beasys.Tobj.FactoryFinder`)
InterfaceRepository	InterfaceRepository object (`org.omg.CORBA.Repository`)
SecurityCurrent	SecurityCurrent object (`org.omg.SecurityLevel2.Current`)
TransactionCurrent	OTS Current object (`com.beasys.Tobj.TransactionCurrent`)

Table 4-3 describes the Tobj module exceptions.

Table 4-3 Tobj Module Exceptions

Exception	Description
`com.beasys.Tobj.InvalidName`	Raised if `id` is not one of the names specified in Table 4-2. On the server, the `resolve_initial_references` method also raises `com.beasys.Tobj.InvalidName` when `SecurityCurrent` is passed.
`com.beasys.Tobj.InvalidDomain`	On the server application, raised if the WebLogic Enterprise server environment is not booted.
`org.omg.CORBA.NO_PERMISSION`	Raised if `id` is `TransactionCurrent` or `SecurityCurrent` and another Bootstrap object in the client owns the Current objects.

Table 4-3 Tobj Module Exceptions

Exception	Description
`org.omg.CORBA.BAD_PARAM`	Raised for the `register_callback_port` method if the object is null or if the host contained in the object does not match the connection.
`org.omg.CORBA.IMP_LIMIT`	Raised if `the register_callback_port` method is invoked more than once.

Java Mapping

Listing 4-1 shows the `Tobj_Bootstrap.java` mapping.

Listing 4-1 Tobj_Bootstrap.java Mapping

```
package com.beasys;

public class Tobj_Bootstrap {
       public Tobj_Bootstrap(org.omg.CORBA.ORB orb,
                               String address)
              throws org.omg.CORBA.SystemException;
public class Tobj_Bootstrap {
      public Tobj_Bootstrap(org.omg.CORBA.ORB orb, String address,
                               java.applet.Applet applet)
              throws org.omg.CORBA.SystemException;

public void register_callback_port(orb.omg.CORBA.Object objref)
               throws org.omg.CORBA.SystemException;

public org.omg.CORBA.Object
                    resolve_initial_references(String id)
              throws Tobj.InvalidName,
                      org.omg.CORBA.SystemException;
public void destroy_current()
              throws org.omg.CORBA.SystemException;
}
```

Programming Examples

This section provides Java client programming examples that use Bootstrap objects.

Getting a SecurityCurrent Object

Listing 4-2 shows how to program a Java client to get a SecurityCurrent object.

Listing 4-2 Programming a Java Client to Get a SecurityCurrent Object

```
import java.util.*;
import org.omg.CORBA.*;
import com.beasys.*;
class client {
      public static void main(String[] args)
      {
              Bool is_native=true;
              Properties = prop;
              Tobj.PrincipalAuthenticator auth = null;
              if (args.length != 1)
                    is_native=false

              if (is_native) {
                    /* Native Client */
                    prop = Tobj_Bootstrap.getNativeProperties();
                    host_port = "";
              } else {
                    /* Remote Client */
                    prop = Tobj_Bootstrap.getRemoteProperties();
                    // Set host and port.
                    host_port = "//COLORMAGIC:10000";
              }
              try {
                    // Initialize ORB
                    ORB orb = ORB.init(args, prop);
                    // Create Bootstrap object
                    Tobj_Bootstrap bs=new Tobj_Bootstrap(orb,host_port);

                    // Get security current
                    org.omg.CORBA.Object ocur =
                    bs.resolve_initial_references("SecurityCurrent");
              SecurityLevel2.Current cur =
                    SecurityLevel2.CurrentHelper.narrow(ocur);
        }
        catch (Tobj.InvalidName e) {
              System.out.println("Invalid name: "+e);
              System.exit(1);
        }
        catch (Tobj.InvalidDomain e) {
           System.out.println("Invalid domain address: "+host_port +" "+e);
              System.exit(1);
        }
        catch (SystemException e) {
           System.out.println("Exception getting security current: "+e);
              e.printStackTrace();
              System.exit(1);
          }
  }
}
```

Getting a UserTransaction Object

The following code example shows using the Bootstrap object to get the UserTransaction object, which may then be used to begin and terminate transactions and get information about transactions.

Listing 4-3   Programming a Java Client to get a UserTransaction Object

```
if (is_native){
      /* Native Client */
      prop = Tobj_Bootstrap.getNativeProperties();
      host_port = null;
} else {
      /* Remote Client */
      prop = Tobj_Bootstrap.getRemoteProperties();
      // Set host and port.
      host_port = "//COLORMAGIC:10000";
}

// Initialize ORB
   orb = ORB.init(args, prop);
```

```
// Create Bootstrap Object
   bs = new Tobj_Bootstrap(orb, host_port);

javax.transaction.UserTransaction ucur = bs.getUserTransaction();

ucur.begin();
/* Make transactional calls from client to server */
   ucur.commit();
```

CHAPTER

5 FactoryFinder Interface

This chapter contains the following topics:

- Capabilities, Limitations, and Requirements
- Functional Description. This section describes the following topics:
  - Locating a FactoryFinder
  - Registering a Factory
  - Locating a Factory
  - Creating Application Factory Keys
- Java Methods
- Java Programming Examples

The FactoryFinder interface provides clients with one object reference that serves as the single point of entry into the WebLogic Enterprise domain. The WebLogic Enterprise NameManager provides the mapping of factory names to object references for the FactoryFinder. Multiple FactoryFinders and NameManagers together provide increased availability and reliability. In this release, the level of functionality has been extended to support multiple domains.

Note: The NameManager is not a naming service, such as CORBAservices Naming Service, but is merely a vehicle for storing registered factories.

In the WebLogic Enterprise environment, application factory objects are used to create objects that clients interact with to perform their business operations (for example, TellerFactory and Teller). Application factories are generally created during server initialization and are accessed by both remote clients and clients located within the server application.

The FactoryFinder interface and the NameManager services are contained in separate (nonapplication) servers. A set of application programming interfaces (APIs) is provided so that both client and server applications can access and update the factory information.

The support for multiple domains in this release benefits customers that need to scale to a large number of machines or who want to partition their application environment. To support multiple domains, the mechanism used to find factories in a WebLogic Enterprise environment has been enhanced to allow factories in one domain to be visible in another. The visibility of factories in other domains is under the control of the system administrator.

Capabilities, Limitations, and Requirements

During server application initialization, application factories need to be registered with the NameManager. Clients can then be provided with the object reference of a FactoryFinder to allow them to retrieve a factory object reference based on associated names that were created when the factory was registered.

The following functional capabilities, limitations, and requirements apply to this release:

- The FactoryFinder interface is in compliance with the `org.omg.CosLifeCycle.FactoryFinder` interface.
- Server applications can register and unregister application factories with the CORBAservices Naming Service.
- Clients can access objects using a single point of entry -- the FactoryFinder.
- Clients can construct names for objects using a simplified BEA scheme made possible by WebLogic Enterprise extensions to the CORBAservices interface or the more general CORBA scheme.
- Multiple FactoryFinders and NameManagers can be used to increase availability and reliability in the event that one FactoryFinder or NameManager should fail.
- Support for multiple domains. Factories in one domain can be configured to be visible in another domain that is under administrative control.

- Two NameManager services, at a minimum, must be configured, preferably on different machines, to maintain the factory-to-object reference mapping across process failures. If both NameManagers fail, the master NameManager, which has been keeping a persistent journal of the registered factories, recovers the previous state by processing the journal so as to re-establish its internal state.
- Only one NameManager must be designated as the master, and the master NameManager must be started before the slave. If the master NameManager is started after one or more slaves, the master assumes that it is in recovery mode instead of in initializing mode.

Functional Description

The WebLogic Enterprise environment promotes the use of the factory design pattern as the primary means for a client to obtain a reference to an object. Through the use of this design pattern, client applications require a mechanism to obtain a reference to an object that acts as a factory for another object. Because the WebLogic Enterprise environment has chosen CORBA as its visible programming model, the mechanism used to locate factories is modeled after the FactoryFinder as described in the CORBAservices Specification, Chapter 6 “Life Cycle Service,” December 1997, published by the Object Management Group.

In the CORBA FactoryFinder model, application servers register active factories with a FactoryFinder. When an application server’s factory becomes inactive, the application server removes the corresponding registration from the FactoryFinder. Client applications locate factories by querying a FactoryFinder. The client application can control the references to the factory object returned by specifying criteria that is used to select one or more references.

Locating a FactoryFinder

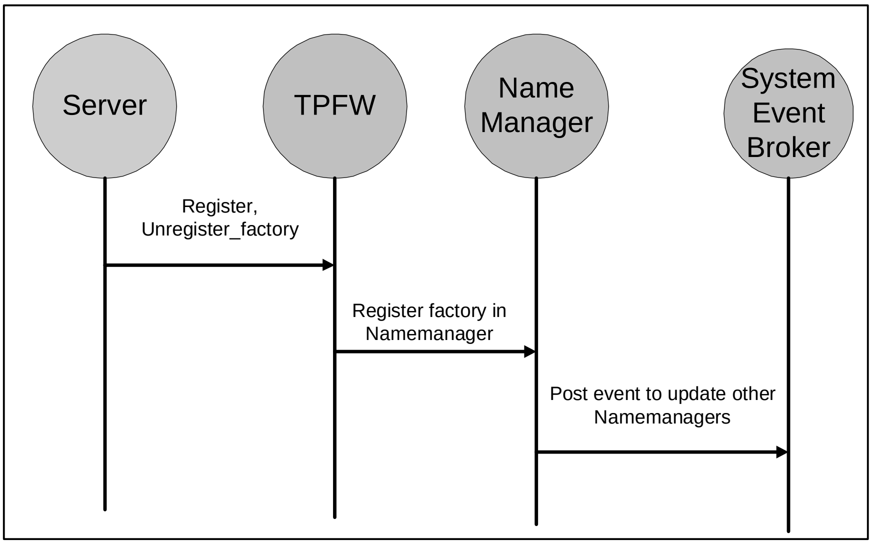
A client application must obtain a reference to a FactoryFinder before it can begin locating an appropriate factory. To obtain a reference to a FactoryFinder in the domain to which a client application is associated, the client application must invoke the `Tobj_Bootstrap.resolve_initial_references` operation with a value of `“FactoryFinder”`. This operation returns a reference to a FactoryFinder that is in the domain to which the client application is currently attached. For more information, see the description of the `com.beasys.Tobj_Bootstrap` object in the *Java API Reference*.

Note: The references to the FactoryFinder that are returned to the client application can be references to factory objects that are registered on the same machine as the FactoryFinder, on a different machine than the FactoryFinder, or possibly in a different domain than the FactoryFinder.

Registering a Factory

For a client application to be able to obtain a reference to a factory, an application server must register a reference to any factory object for which it provides an implementation with the FactoryFinder (see Figure 5-1). Using the WebLogic Enterprise TP Framework, the registration of the reference for the factory object can be accomplished using the `TP.register_factory` operation, once a reference to a factory object has been created. The reference to the factory object, along with a value that identifies the factory, is passed to this operation. The registration of references to factory objects is typically done as part of initialization of the application; normally, as part of the implementation of the `Server.initialize` operation.

Figure 5-1 Registering a Factory Object

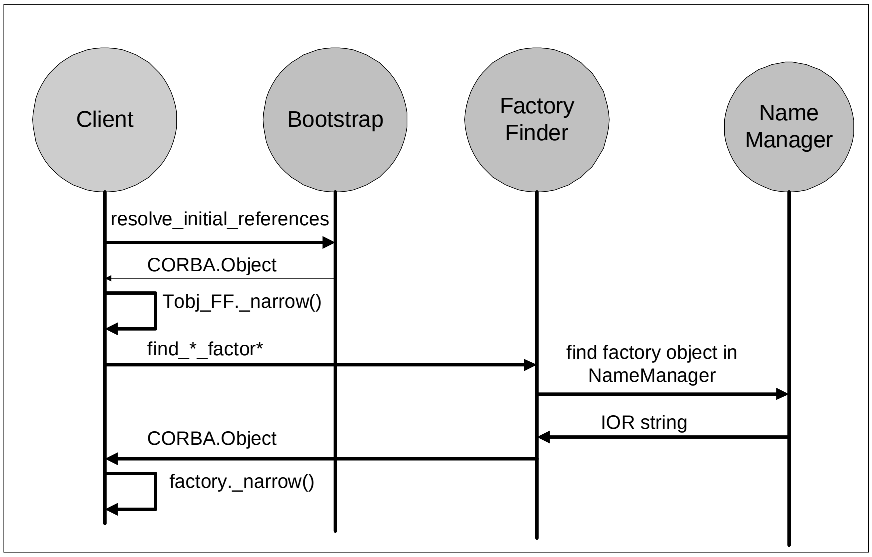

Server
TPFW
Name Manager
System Event Broker
Register, Unregister_factory
Register factory in Namemanager
Post event to update other Namemanagers

When the server application is shutting down, it must unregister any references to the factory object that it has previously registered in the application server. This is done by passing the same reference to the factory object, along with the corresponding value used to identify the factory, to the `TP.unregister_factory` operation. Once unregistered, the reference to the factory object can then be destroyed. The process of unregistering a factory with the FactoryFinder is typically done as part of the implementation of the `Server.release` operation. For more information about these operations, see the section “Server Interface” on page 3-23.

Locating a Factory

For a client application to request a factory to create a reference to an object, it must first obtain a reference to the factory object. The reference to the factory object is obtained by querying a FactoryFinder with specific selection criteria, as shown in Figure 5-2. The criteria are determined by the format of the particular FactoryFinder interface and method used.

Figure 5-2 Locating a Factory Object



The WebLogic Enterprise software extends the `CosLifeCycle.FactoryFinder` interface by introducing three methods in addition to the `find_factories` method declared for the FactoryFinder. Therefore, using the Tobj extensions, a client can use either the `find_factories` or `find_factories_by_id` methods to obtain a list of application factories. A client can also use the `find_one_factory` or `find_one_factory_by_id` method to obtain a single application factory, and the `list_factories` method to obtain a list of all registered factories.

The `CosLifeCycle.FactoryFinder` interface defines a `factory_key`, which is a sequence of `id` and `kind` strings conforming to the CosNaming Name shown in Listing 5-1. The `kind` field of the NameComponent for all WebLogic Enterprise application factories is set to the string `FactoryInterface` by the TP Framework when an application factory is registered. Applications supply their own value for the `id` field.

Assuming that the CORBAservices Life Cycle Service modules are contained in their own file (`ns.idl` and `lcs.idl`, respectively), only the OMG IDL code for that subset of both files that is relevant for using the WebLogic Enterprise FactoryFinder is shown in the following listings.

CORBAservices Naming Service Module OMG IDL

Listing 5-1 shows the portions of the `ns.idl` file that are relevant to the FactoryFinder.

Listing 5-1 CORBAservices Naming OMG IDL

```
// ------  ns.idl  ------

module CosNaming {
       typedef string Istring;
       struct NameComponent {
              Istring id;
              Istring kind;
       };
       typedef sequence <NameComponent> Name;

};

// This information is taken from CORBAservices: Common Object
// Services Specification, page 3-6. Revised Edition:
// March 31, 1995. Updated: November 1997. Used with permission by
OMG.
```

CORBAservices Life Cycle Service Module OMG IDL

Listing 5-2 shows the portions of the `lcs.idl` file that are relevant to the FactoryFinder.

Listing 5-2 Life Cycle Service OMG IDL

```
// -----  lcs.idl  -----

#include “ns.idl”

module CosLifeCycle{
       typedef CosNaming::Name Key;
       typedef Object Factory;
        typedef sequence<Factory> Factories;

       exception NoFactory{ Key search_key; }
```

```
        interface FactoryFinder {
             Factories find_factories(in Key factory_key)
                   raises(NoFactory);

         };

};

// This information is taken from CORBAservices: Common Object
// Services Specification, pages 6-10, 11. Revised Edition:
// March 31, 1995. Updated: November 1997. Used with permission by
OMG.
```

Tobj Module OMG IDL

Listing 5-3 shows the Tobj Module OMG IDL.

Listing 5-3 Tobj Module OMG IDL

```
// -----  Tobj.idl  -----

module Tobj {

   // Constants

   const string FACTORY_KIND = "FactoryInterface";

   // Exceptions

   exception CannotProceed { };
   exception InvalidDomain {};
   exception InvalidName { };
   exception RegistrarNotAvailable { };

   // Extension to LifeCycle Service

   struct FactoryComponent {
        CosLifeCycle::Key factory_key;
        CosLifeCycle::Factory factory_ior;
   };

   typedef sequence<FactoryComponent> FactoryListing;

   interface FactoryFinder : CosLifeCycle::FactoryFinder {
     CosLifeCycle::Factory find_one_factory(in CosLifeCycle::Key
                                             factory_key)
```

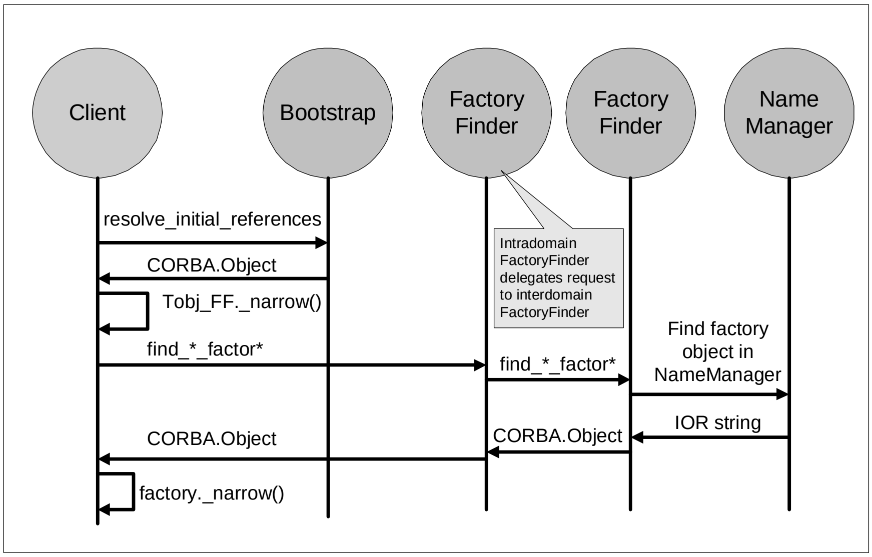
```
                    raises (CosLifeCycle::NoFactory,
                            CannotProceed,
                            RegistrarNotAvailable);
        CosLifeCycle::Factory find_one_factory_by_id(in string
                                                     factory_id)
                     raises (CosLifeCycle::NoFactory,
                             CannotProceed,
                             RegistrarNotAvailable);
        CosLifeCycle::Factories find_factories_by_id(in string
                                                     factory_id)
                       raises (CosLifeCycle::NoFactory,
                               CannotProceed,
                               RegistrarNotAvailable);
        FactoryListing list_factories()
                         raises (CannotProceed,
                                 RegistrarNotAvailable);
   };
};
```

Locating Factories in Another Domain

Typically, a FactoryFinder returns references to factory objects that are in the same domain as the FactoryFinder itself. However, it is possible to return references to factory objects in domains other than the domain in which a FactoryFinder exists. This can occur if a FactoryFinder contains information about factories that are resident in another domain (see Figure 5-3). A FactoryFinder finds out about these interdomain factory objects through configuration information that describes the location of these other factory objects.

When a FactoryFinder receives a request to locate a factory object, it must first determine if a reference to a factory object that meets the specified criteria exists. If there is registration information for a factory object that matches the criteria, the FactoryFinder must then determine if the factory object is local to the current domain or needs to be imported from another domain. If the factory object is from the local domain, the FactoryFinder returns the reference to the factory object to the client.

Figure 5-3 Inter-domain FactoryFinder Interaction



If, on the other hand, the information indicates that the factory object is from another domain, the FactoryFinder delegates the request to an interdomain FactoryFinder in the appropriate domain. As a result, only a FactoryFinder in the same domain as the factory object will contain a reference to the factory object. The interdomain FactoryFinder is responsible for returning the reference of the factory object to the local FactoryFinder, which subsequently returns it to the client.

Why Use WebLogic Enterprise Extensions?

The WebLogic Enterprise software extends the interfaces defined in the CORBAservices specification, Chapter 6 “Life Cycle Service,” December 1997, published by the Object Management Group, for the following reasons:

- Although the CORBA-defined approach is powerful and allows various selection criteria, the interface used to query a FactoryFinder can be complicated to use.
- Additionally, if the selection criterion specified by the client application is not specific enough, it is possible that more than one reference to a factory object may be returned. If this occurs, it is not immediately obvious what a client application should do next.

- Finally, the CORBAservices specification did not specify a standardized mechanism through which an application server is to register a factory object.

Therefore, WebLogic Enterprise extends the interfaces defined in the CORBAservices specification to make using a FactoryFinder easier. The extensions are manifested as refined interfaces to the FactoryFinder that are derived from the interfaces specified in the CORBAservices specification.

Creating Application Factory Keys

Two of the four methods provided in the `Tobj.FactoryFinder` interface accept `CosLifeCycle.Keys`, which corresponds to `CosNaming.Name`. A client must be able to construct these keys.

The CosNaming Specification describes two interfaces that constitute a Names Library interface that can be used to create and manipulate `CosLifeCycle.Keys`. The pseudo OMG IDL statements for these interfaces is described in the following section.

Names Library Interface Pseudo OMG IDL

Note: This information is taken from the *CORBAservices: Common Object Services Specification*, pp. 3-14 to18. Revised Edition: March 31, 1995. Updated: November 1997. Used with permission by OMG.

To allow the representation of names to evolve without affecting existing client applications, it is desirable to hide the representation of names from the client application. Ideally, names themselves would be objects; however, names must be lightweight entities that are efficient to create, manipulate, and transmit. As such, names are presented to programs through the names library.

The names library implements names as pseudo-objects. A client application makes calls on a pseudo-object in the same way it makes calls on an ordinary object. Library names are described in pseudo-IDL (to suggest the appropriate language binding). C++ client applications use the same client language bindings for pseudo-IDL (PIDL) as they use for IDL.

Pseudo-object references cannot be passed across OMG IDL interfaces. As described in Chapter 3 of the *CORBAservices: Common Object Services Specification*, in the section “The CosNaming Module,” the CORBAservices Naming Service supports the

NamingContext OMG IDL interface. The names library supports an operation to convert a library name into a value that can be passed to the name service through the NamingContext interface.

Note: It is not a requirement to use the names library in order to use the CORBAservices Naming Service.

The names library consists of two pseudo-IDL interfaces, the LNameComponent interface and the LName interface, as shown in Listing 5-4.

Listing 5-4 Names Library Interfaces in Pseudo-IDL

```
interface LNameComponent { //  PIDL
    const short MAX_LNAME_STRLEN = 128;

    exception NotSet{ };
    exception OverFlow{ };

    string get_id
           raises (NotSet);
    void set_id(in string i)
           raises (OverFlow);
    string get_kind()
           raises(NotSet);
    void set_kind(in string k)
           raises (OverFlow);
    void destroy();
};

interface LName {// PIDL
       exception NoComponent{ };
       exception OverFlow{ };
       exception InvalidName{ };
       LName insert_component(in unsigned long i,
                    in LNameComponent n)
              raises (NoComponent, OverFlow);
       LNameComponent get_component(in unsigned long i)
              raises (NoComponent);
       LNameComponent delete_component(in unsigned long i)
              raises (NoComponent);
       unsigned long num_components();
       boolean equal(in LName ln);
       boolean less_than(in LName ln);
       Name to_idl_form()
              raises (InvalidName);
       void from_idl_form(in Name n);
```

```
      void destroy();
};

LName create_lname();
LNameComponent create_lname_component();
```

Creating a Library Name Component

To create a library name component pseudo-object, use the following method:

```
LNameComponent create_lname_component();
```

The returned pseudo-object can then be operated on using the operations shown in Listing 5-4.

Creating a Library Name

To create a library name pseudo-object, use the following method:

```
LName create_lname();
```

The returned pseudo-object reference can then be operated on using the operations shown in Listing 5-4.

The LNameComponent Interface

A name component consists of two attributes: `identifier` and `kind`. The LNameComponent interface defines the operations associated with these attributes, as follows:

```
string get_id()
raises(NotSet);
void set_id(in string k);
string get_kind()
raises(NotSet);
void set_kind(in string k);
```

`get_id`

The `get_id` operation returns the `identifier` attribute's value. If the attribute has not been set, the `NotSet` exception is raised.

`set_id`

The `set_id` operation sets the `identifier` attribute to the string argument.

`get_kind`

The `get_kind` operation returns the `kind` attribute's value. If the attribute has not been set, the `NotSet` exception is raised.

`set_kind`

The `set_kind` operation sets the `kind` attribute to the string argument.

The LName Interface

The following operations are described in this section:

- Destroying a library name component pseudo-object
- Inserting a name component
- Getting the ith name component
- Deleting a name component
- Number of name components
- Testing for equality
- Testing for order
- Producing an OMG IDL form
- Translating an OMG IDL form
- Destroying a library name pseudo-object

Destroying a Library Name Component Pseudo-Object

The destroy operation destroys library name component pseudo-objects.

```
void destroy();
```

Inserting a Name Component

A name has one or more components. Each component except the last is used to identify names of subcontexts. (The last component denotes the bound object.) The `insert_component` operation inserts a component after position `i`.

```
LName insert_component(in unsigned long i, in LNameComponent lnc)
raises(NoComponent, OverFlow);
```

If component `i-1` is undefined and component `i` is greater than 1 (one), the `insert_component` operation raises the `NoComponent` exception.

If the library cannot allocate resources for the inserted component, the `OverFlow` exception is raised.

Getting the ith Name Component

The `get_component` operation returns the `i`th component. The first component is numbered 1 (one).

```
LNameComponent get_component(in unsigned long i)
raises(NoComponent);
```

If the component does not exist, the `NoComponent` exception is raised.

Deleting a Name Component

The `delete_component` operation removes and returns the `i`th component.

```
LNameComponent delete_component(in unsigned long i)
     raises(NoComponent);
```

If the component does not exist, the `NoComponent` exception is raised.

After a `delete_component` operation has been performed, the compound name has one fewer component and components previously identified as `i+1...n` are now identified as `i...n-1`.

Number of Name Components

The `num_components` operation returns the number of components in a library name.

```
unsigned long num_components();
```

Testing for Equality

The equal operation tests for equality with library name `ln`.

```
boolean equal(in LName ln);
```

Testing for Order

The `less_than` operation tests for the order of a library name in relation to library name `ln`.

```
boolean less_than(in LName ln);
```

This operation returns true if the library name is less than the library name `ln` passed as an argument. The library implementation defines the ordering on names.

Producing an OMG IDL form

Pseudo-objects cannot be passed across OMG IDL interfaces. The library name is a pseudo-object; therefore, it cannot be passed across the OMG IDL interface for the CORBAservices Naming Service. Several operations in the NamingContext interface have arguments of an OMG IDL-defined structure, `Name`. The following PIDL operation on library names produces a structure that can be passed across the OMG IDL request.

```
Name to_idl_form()
    raises(InvalidName);
```

If the name is of length 0 (zero), the `InvalidName` exception is returned.

Translating an IDL Form

Pseudo-objects cannot be passed across OMG IDL interfaces. The library name is a pseudo-object; therefore, it cannot be passed across the OMG IDL interface for the CORBAservices Naming Service. The NamingContext interface defines operations that return an IDL struct of type `Name`. The following PIDL operation on library names sets the components and `kind` attribute for a library name from a returned OMG IDL defined structure, `Name`.

```
void from_idl_form(in Name n);
```

Destroying a Library Name Pseudo-Object

The `destroy` operation destroys library name pseudo-objects.

```
void destroy();
```

Java Mapping

The names library pseudo OMG IDL interface maps to the Java classes contained in the `com.beasys.Tobj` package, shown in Listing 5-5. All exceptions are contained in the same package.

For a detailed description of the Library Name class, refer to Chapter 3 in the *CORBAservices: Common Object Services Specification*.

Listing 5-5 Java Mapping for LNameComponent

```
package com.beasys.Tobj;

public class LNameComponent {
    public static LNameComponent create_lname_component();
    public static final short MAX_LNAME_STRING = 128;
    public void destroy();
    public String get_id() throws NotSet;
    public void   set_id(String i) throws OverFlow;
    public String get_kind() throws NotSet;
    public void   set_kind(String k) throws OverFlow;
};

package com.beasys.Tobj;

public class LName {

   public static LName create_lname();
   public void destroy();
   public LName insert_component(long i, LNameComponent n)
     throws NoComponent, OverFlow;
   public LNameComponent get_component(long i)
     throws NoComponent;
   public LNameComponent delete_component(long i)
     throws NoComponent;
   public long num_components();
   public boolean equal(LName ln);
   public boolean less_than(LName ln);// not implemented
   public org.omg.CosNaming.NameComponent[] to_idl_form()
     throws InvalidName;
  public void from_idl_form(org.omg.CosNaming.NameComponent[] nr);
};
```

Java Methods

The documentation for the Java methods on the FactoryFinder interface is in the *Java API Reference*.

Java Programming Examples

The following listings show Java programming examples of how to program using the FactoryFinder interface.

Note: Remember to check for exceptions in your code.

Server Registering a Factory

Listing 5-6 shows how to program a server to register a factory.

Listing 5-6 Server Application: Registering a Factory

```
// Register the factory reference with the factory finder.
//
// The second parameter to TP.register_factory() is a string
// identifier that is used to identify the object.
// This same string is used in the call to TP.unregister_factory().
// It is also used in the call to find_one_factory_by_id() that
// is called by clients of this interface.
//
TP.register_factory(
    fact_oref,      // factory object reference
    tellerFName     // factory name
    );
```

Client Obtaining a FactoryFinder Object Reference

Listing 5-7 shows how to program a client to get a FactoryFinder object reference.

Listing 5-7 Client Application: Getting a FactoryFinder Object Reference

```
// Create the Bootstrap object,
// the TOBJADDR properly contains host and port to connect to.
Tobj_Bootstrap bootstrap = new Tobj_Bootstrap (orb,"");

// Use the Bootstrap object to find the factory finder.
org.omg.CORBA.Object fact_finder_oref =
    bootstrap.resolve_initial_references("FactoryFinder");

// Narrow the factory finder.
FactoryFinder fact_finder_ref =
    FactoryFinderHelper.narrow(fact_finder_oref);
```

Client Finding One Factory Using the Tobj Approach

Listing 5-8 shows how to program a client to find one factory using the Tobj approach.

Listing 5-8 Client Application: Finding One Factory Using the Tobj Approach

```
// Use the factory finder to find the teller factory.
org.omg.CORBA.Object teller_fact_oref =
fact_finder_ref.find_one_factory_by_id("TellerFactory_1");
```

CHAPTER

6 Security Service

This chapter contains the following topics:

- Introduction
- Capabilities and Limitations
- Getting Initial References to the SecurityCurrent Object
- Basic Security-Level Requirements for WebLogic Enterprise Clients
- Functional Components. This section includes the following topics:
  - Security Model
  - Authentication of Principals
  - Controlling Access to Objects
  - Administrative Control
- Security Model Functional Description
- Authentication
- Client Security API. This section describes the following modules:
  - CORBA Module
  - TimeBase Module
  - Security Module
  - Security Level 1 Module
  - Security Level 2 Module
  - Tobj Module
- Java Programming Examples

Introduction

The purpose of client security is to enable WLE clients to authenticate themselves via the IIOP Server Listener/Handler and to pass the WLE security checks.

WLE client security provides two types of security authentication:

- An implementation of the security environmental objects for a CORBA Object Request Broker (ORB) environment. Client authentication is achieved using application programming interfaces (APIs) defined by CORBA security, although the authentication is performed by the IIOP Server Listener/Handler, not by the client ORB. Client security provides helper methods to create the data structures needed to call the standard CORBA authentication methods.

- An implementation of authentication similar to that found in the BEA TUXEDO system. Logon and logoff functions are provided that are easier to use than their CORBA counterparts. Logon passwords and data are secure traversing the network.

You can use either method to implement client security.

Capabilities and Limitations

This implementation of WebLogic Enterprise client security has the following capabilities and limitations.

- Supports two types of authentication, as described above.
- Provides add-on methods to help generate the information needed for CORBA security from information specific to the WebLogic Enterprise client, such as client name, client application password, user password (or user authentication data), and so forth.
- Implements authentication only.
- Allows remote WebLogic Enterprise clients to authenticate themselves to WebLogic Enterprise domains via the IIOP Server Listener/Handler so that clients can connect to a WebLogic Enterprise domain with BEA TUXEDO style security activated.

Getting Initial References to the SecurityCurrent Object

You use the Bootstrap object to get an initial reference to the SecurityCurrent object. For a description of the Bootstrap object method, refer to the `com.beasys.Tobj_Bootstrap.resolve_initial_references` method description in the *Java API Reference*.

Basic Security-Level Requirements for WebLogic Enterprise Clients

A client that connects to a WebLogic Enterprise domain must provide security information according to the security level required by the WebLogic Enterprise domain. Table 6-1 defines the security levels supported by WebLogic Enterprise domains.

Table 6-1 Security Levels Supported by WebLogic Enterprise Domains

Security Level	Definition
`TOBJ_NOAUTH`	No authentication is needed; however, the client can still authenticate itself, and must specify a user name and a client name, but no password.
`TOBJ_SYSAUTH`	The client must authenticate itself to the WebLogic Enterprise domain, and must specify a user name, client name, and client application password.
`TOBJ_APPAUTH`	The client must provide information in addition to that which is required by `TOBJ_SYSAUTH`. If the default WebLogic Enterprise authentication service is used in the WebLogic Enterprise domain configuration, the client must provide a user password; otherwise, the client provides authentication data that is interpreted by the custom authentication service in the WebLogic Enterprise domain.

Functional Components

This section describes the functional components of the Security Service.

Security Model

The security model in the WebLogic Enterprise software defines the overall framework for security. The WebLogic Enterprise product provides the flexibility to support different security mechanisms and policies that can be used to achieve the appropriate level of functionality and assurance.

The WebLogic Enterprise security model defines:

- Under what conditions clients may access objects
- What authentication of users and other principals is required, who they are, and what they can do

The WebLogic Enterprise security model is a combination of the security reference model defined in the CORBAservices Security Service specification[1]and the value-added extensions that provide a focused, simplified form of the security found in BEA TUXEDO. The WebLogic Enterprise security model allows application developers to choose to use the security model defined by CORBA, or the BEA extensions, when developing an application.

Authentication of Principals

Authentication of principals, typically a human user or system entity, provides security officers with the ability to ensure that only registered principals have access to the objects in the system. An authenticated principal is used as the primary mechanism to control access to objects.

1. All references to the CORBAservices Security Service specification in this document are to the Revision 1.5, December 1998 edition, published by the Object Management Group.

The act of authenticating principals allows the security mechanisms to:

- Make principals accountable for their actions
- Control access to protected objects
- Identify the originator of a request

Controlling Access to Objects

The WebLogic Enterprise security model provides a simple framework through which a security officer can limit access to authorized users only. Limiting access to objects allows security officers to prohibit access to objects by unauthorized principals.

The access control framework consists of two parts:

- The object invocation policy that is enforced automatically on object invocation
- An application access policy that the user-written application can enforce itself

Administrative Control

The system administrator is responsible for setting security policies for client machines, server machines, and IIOP Listener/Handlers that interact with applications in their WebLogic Enterprise domain. While the administrator sets the general policies, another person or group of people may be responsible for managing security (users, permissions, and so forth).

To provide system administrators the ability to define and enforce authentication of the principal, the software provides a set of configuration parameters and utilities. Through these features, a system administrator can configure the WebLogic Enterprise software to force the principals to be authenticated to access a system on which WebLogic Enterprise software is installed.

Security Model Functional Description

This section provides a functional description of the security model.

Description

The security model adopted in the WebLogic Enterprise software is based largely on the CORBA security model defined in the CORBAservices Security Service specification. Consequently, many of the concepts found in the CORBA security model apply to WebLogic Enterprise security.

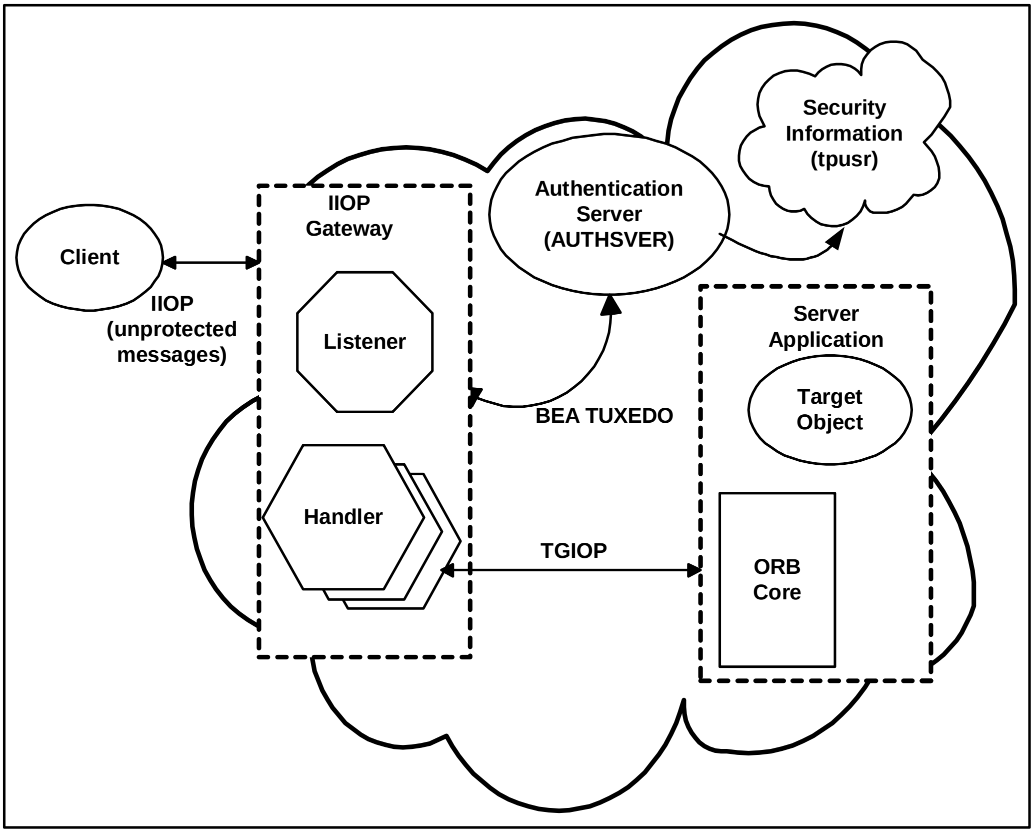
In addition to many of the interfaces defined by the CORBAservices Security Service, BEA provides extensions, in the form of derived interfaces. These extensions expose the security functionality found in the BEA TUXEDO system as CORBA interfaces that are found in the `Tobj` namespace. The interfaces in the `Tobj` namespace are intended to be familiar to developers of BEA TUXEDO applications and provide a focused, simplified form of the equivalent CORBA-defined capability. An application developer can choose to use the CORBA-defined security model, or the BEA extensions, when developing an application.

In a security model, there are usually defined sets of specific security policies. The WebLogic Enterprise security model defines policies that specify whether a principal must be authenticated to use the system.

WebLogic Enterprise implements a delegated trust authentication model. In this model, clients authenticate to a trusted system gateway process. In the case of WebLogic Enterprise, the trusted system gateway process is the ISL/ISH. As part of a successful authentication, a security association (called a security context) is established between the client application and the ISL/ISH that is used to mediate access to objects. The WebLogic Enterprise software associates the security context with the network connection over which the principal was authenticated. Except for the authentication exchange, this is currently the default behavior of the WebLogic Enterprise system.

Figure 6-1 shows the security environment components.

Figure 6-1 WebLogic Enterprise Security Environment


Security Information (tpusr)
Authentication Server (AUTHSVER)
IIOP Gateway
Client
IIOP (unprotected messages)
Listener
BEA TUXEDO
Server Application
Target Object
Handler
TGIOP
ORB Core

Logging on to the System

When a user or other principal wants to use the WebLogic Enterprise system, the principal usually needs to authenticate and obtain credentials. The credentials obtained by the principal contain identity attributes that are used to control access to WebLogic Enterprise servers.

The WebLogic Enterprise Principal Authentication object provides a delegation mechanism to provide security to non-BEA branded clients. As illustrated in Figure 6-1, remote client applications perform authentication with the ISL/ISH, instead of with the server application itself. Consequently, the establishment of a security association is performed in the ISL/ISH, rather than in the server-side ORB.

In terms of CORBA security, the ISL/ISH acts as a CORBA-defined half-gateway into the WebLogic Enterprise domain, and is, therefore, responsible for providing the security mechanisms that will be used in secure invocations for a given object.

Example of a Secure Object Invocation

The following is a description of what happens when a client invokes on a target object in a WebLogic Enterprise environment:

- The client application obtains credentials for the user by authenticating itself with the WebLogic Enterprise domain using a PrincipalAuthenticator object. The request for authentication is sent to the ISL/ISH that relays the requests to an authentication server, which verifies the supplied information. If the verification process succeeds, the security system constructs a Credentials object that is used in all future invocations. The Credentials object for the principal is associated with the SecurityCurrent object that represents the security context for the current thread of execution.
- The client application invokes an object in the domain using its object reference. The request is packaged into an IIOP request and forwarded to the ISL/ISH that associates the request with the previously established security association. At this point, the ISL/ISH forwards the request, along with the credentials of the initiating principal, to an appropriate server process.

Authentication

When an active entity wants to use a secure object system, it authenticates itself and obtains credentials. The credentials contain its certified identity, and, optionally, its privilege attributes, and control where and when they can be used. In the WebLogic Enterprise security model, an active entity must establish its rights to access objects in the system. The active entity must be either a principal, or a client that is acting on behalf of a principal.

A principal is defined to be either a user or a system entity that is registered in and authenticatable to the security system. Authentication may be accomplished in a number of ways. The most common way is for a user to supply a password. When a user or other principal is authenticated, the principal usually supplies:

- The principal’s security name
- The authentication information needed by the particular authentication method used

Once authenticated, the principal’s security attributes are maintained in the security system in a credential. The credentials provide a means for the security system to provide the principal’s certified identity and describes the privileges granted to the particular principal.

Principals who initiate activities, have one identity that may be used. The identity is represented in the system as attributes.

Authentication Mechanisms

As stated in “Logging on to the System” on page 6-7, the lack of interoperable security amongst the ORB vendors has resulted in it being necessary to utilize a delegation mechanism to provide authentication to client environments. The delegation mechanism used is similar to the mechanism found in BEA TUXEDO. Consequently, an authentication mechanism known as BEA TUXEDO-based security is supported in WebLogic Enterprise domains. The implementation of this mechanism is layered on top of the security mechanism provided in BEA TUXEDO.

As in BEA TUXEDO, remote client applications perform authentication with the ISL/ISH instead of with the server application itself. Consequently, the establishment of a security association is actually performed in the ISL/ISH, rather than in the server-side ORB. Machines and server applications within a WebLogic Enterprise domain are considered trusted. This trust is a result of a defined trust model that is based on the assumption that the machines and applications that make up the domain are under the control of administrators only.

Authentication of principals in an environment based on BEA TUXEDO requires the use of user names and passwords. Unlike most operating systems, BEA TUXEDO security defines three different authentication levels:

- TOBJ_NOAUTH -- no authentication is needed.
- TOBJ_SYSAUTH -- the principal must authenticate itself to the domain, and specify a user name, client name, and user password.
- TOBJ_APPAUTH -- same as TOBJ_SYSAUTH, except that the principal must provide information. If the default authentication service provided in the WebLogic Enterprise software is configured, the principal must provide an

application password; otherwise, the principal provides authentication data that is interpreted by a custom authentication service.

The level of authentication required is administratively controlled and is defined in the application's configuration. Because a client application is typically unaware of the level of authentication configured, it must query the security system to determine the authentication level required.

The configuration of the authentication level required is specified in an application's configuration, not on an object or method level. Consequently, if an application is configured to require authentication, all objects in the application require certified credentials for the principals. Applications can be configured to support either unauthenticated or authenticated principals. In unauthenticated scenarios, application developers may use a Principal Authenticator to assert a user name and client name, neither of which will be verified.

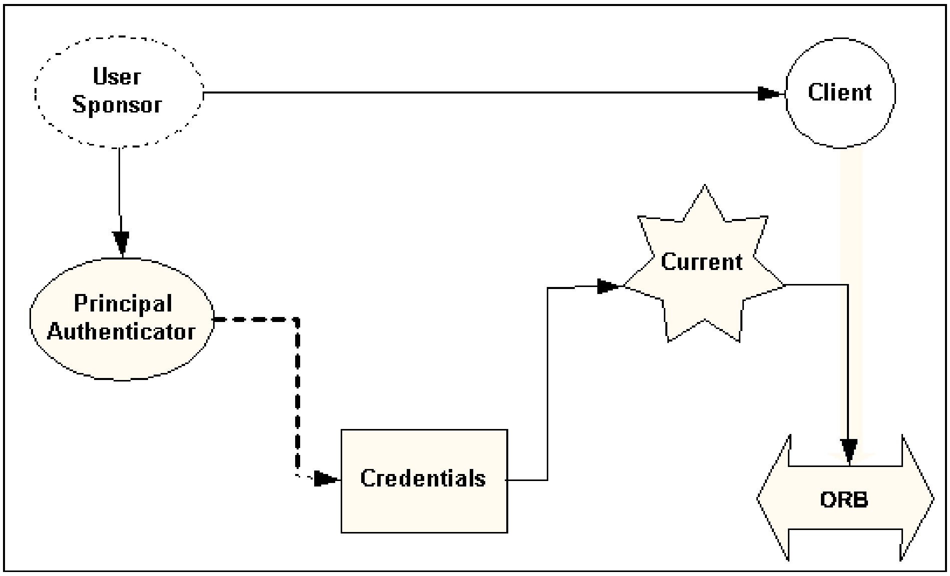
Because the BEA TUXEDO-based authentication mechanism is layered on top of the security mechanisms provided in BEA TUXEDO, it is possible for customers to replace the Authentication Server that provides the default authentication mechanism with a custom implementation. A description of how to replace the Authentication Server in BEA TUXEDO is described in the BEA TUXEDO manuals and is outside the scope of this document.

Authentication Process

The process of authenticating a principal is done by a user sponsor (see Figure 6-2). A user sponsor is the code that calls the security interfaces for user authentication. In a WebLogic Enterprise domain configured to use BEA TUXEDO-based security, the client application is the user sponsor.

In either case, the user provides identity and authentication data, such as a password, to the user sponsor. The user sponsor uses the Principal Authenticator object provided as part of the security implementation to make the calls necessary to authenticate the principal. The credentials for the authenticated principal are associated with the security system's implementation of the SecurityCurrent object and are represented by a Credentials object.

Figure 6-2 Authentication



PRINCIPAL AUTHENTICATOR OBJECT

The Principal Authenticator object is the object visible to the application that is responsible for the creation of Credentials for a given principal. A user or principal that requires authentication but has not been authenticated uses the Principal Authenticator object.

CREDENTIALS OBJECT

A Credentials object holds the security attributes of a principal. These security attributes include its authenticated or unauthenticated identities. It also contains information for establishing security associations. The Credentials object provides methods to obtain security attributes of the principals it represents.

SecurityCurrent Object

The SecurityCurrent object represents the current execution context at the client and target object. The SecurityCurrent object provides methods to give access to security information associated with the execution context. The SecurityCurrent object gives access to the Credentials associated with the execution environment.

At any stage, a client or target object can find the default credentials for subsequent invocations by calling the `Current.get_credentials` method to request the invocation credentials. These default credentials are used in all invocations that use object references.

Principal Authenticator Object

The Principal Authenticator object is used by a user or principal that requires authentication but has not been authenticated prior to calling the object system. The act of authenticating a principal results in the creation of a Credentials object that is made available as the default credentials for the application. The Credentials object is returned so it can be used for other methods on the Credentials.

The Principal Authenticator object is a singleton object; there is only a single instance allowed in a process address space. Multiple references to the Principal Authenticator object must be supported. The Principal Authenticator object is also stateless. A Credentials object is not associated with the Principal Authenticator object that created it.

All Principal Authenticator objects support the `SecurityLevel2.PrincipalAuthenticator` interface defined in the CORBAservices Security Service specification. This interface contains two methods that are used to accomplish the authentication of the principal. This is because authentication of principals may require more than one step. The authenticate method allows the caller to authenticate, and optionally select, attributes for the principal of this session.

Any invocation that fails because the security infrastructure does not permit that invocation raises the standard exception `CORBA.NO_PERMISSION`. A method that fails because the feature requested is not supported by the security infrastructure implementation raises the `CORBA.NO_IMPLEMENT` standard exception. Any parameter that has inappropriate values raises the `CORBA.BAD_PARAM` standard exception.

The Principal Authenticator object is a locality-constrained object. Therefore, a Principal Authenticator object may not be used through the DII/DSI facilities of CORBA. Any attempt to pass a reference to this object outside of the current process, or any attempt to externalize it using `CORBA.ORB.object_to_string`, results in the raising of the `CORBA.MARSHAL` exception.

WebLogic Enterprise Extensions to the Principal Authenticator Object

BEA extends the Principal Authenticator object with functionality to support similar security to that found in BEA TUXEDO. The enhanced functionality is provided by defining the `com.beasys.Tobj.PrincipalAuthenticator` interface. This interface contains methods to provide similar capability to that available from BEA TUXEDO through the `tpinit` function.

The methods defined for the `Tobj.PrincipalAuthenticator` interface are intended to be familiar to developers of BEA TUXEDO applications, and provide a focused, simplified form of the equivalent CORBA-defined capability. An application developer can choose to use the CORBA-defined or BEA extensions when developing an application. The interface `Tobj.PrincipalAuthenticator` is derived from the CORBA `SecurityLevel2.PrincipalAuthenticator` interface.

The extended Principal Authenticator object adheres to all the same rules as the Principal Authenticator object defined in the CORBAservices Security Service specification.

The implementation of the extended Principal Authenticator object requires users to supply a user name, client name, and additional authentication data (for example, passwords) used for authentication. Because the information needs to be transmitted over the network to the ISL/ISH, it is protected to ensure confidentiality. The protection must include encryption of any information provided by the user.

An extended Principal Authenticator object that supports the `Tobj.PrincipalAuthenticator` interface provides the same functionality as if the `SecurityLevel2.PrincipalAuthenticator` interface were used to perform the authentication of the principal. However, unlike the `SecurityLevel2.PrincipalAuthenticator.authenticate` method, the logon method defined on the `Tobj.PrincipalAuthenticator` interface does not return a Credentials object. As a result, multithreaded applications that need to use more than one principal identity are required to call the `Current.get_credentials` method immediately after the logon method to retrieve the Credentials object as a result of the logon method. Retrieval of the Credentials object directly after a logon method should be protected with serialized access since it is possible for another thread to also perform a `logon` method.

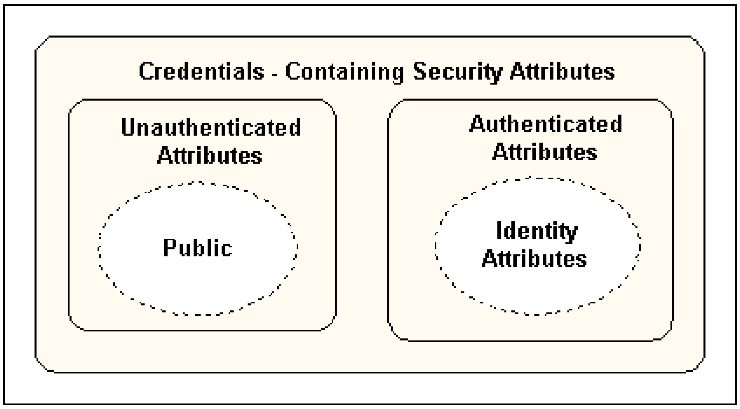
Credentials Object

A Credentials object (see Figure 6-3) holds the security attributes of a principal. The Credentials object provides methods to obtain and set the security attributes of the principals it represents. These security attributes include its authenticated or unauthenticated identities and privileges. The Credentials object also contains information for establishing security associations.

Credentials objects are created as the result of:

- Authentication
- Copying an existing Credentials object
- Asking for a Credentials object via the SecurityCurrent object

Figure 6-3 Credentials Object


Credentials - Containing Security Attributes
Unauthenticated Attributes
Public
Authenticated Attributes
Identity Attributes

There can be more than one Credentials object in a process address space. Multiple references to a Credentials object must be supported. A Credentials object is stateful. It maintains state on behalf of the principal for which it was created. This state includes any information necessary to determine the identity and privileges of the principal it represents. Credentials objects are not associated with the Principal Authenticator object that created it, but must contain some indication of the authentication authority that certified the principal’s identity.

All Credentials objects support the `SecurityLevel2.Credentials` interface. Any invocation that fails as a result of the security infrastructure determining that the client does not have permission, raises the standard exception `CORBA.NO_PERMISSION`. A method that fails because the feature requested is not supported by the security infrastructure implementation raises the `CORBA.NO_IMPLEMENT` standard exception. Any parameter that has inappropriate values raises the `CORBA.BAD_PARAM` standard exception.

The Credentials object is a locality-constrained object. Therefore, a Credentials object may not be used through the DII/DSI facilities. Any attempt to pass a reference to this object outside of the current process, or any attempt to externalize it using `CORBA.ORB.object_to_string`, results in the raising of the `CORBA.MARSHAL` exception.

SecurityCurrent Object

The SecurityCurrent object (see Figure 6-4) represents the current execution context at both client and target objects. The SecurityCurrent object represents service-specific state information associated with the current execution context. Both clients and servers have SecurityCurrent objects that represent state associated with the thread of execution and the capsule (process) in which the thread is executing (their execution contexts).

The SecurityCurrent object is a singleton object; there is only a single instance allowed in a process address space. Multiple references to the SecurityCurrentobject must be supported.

The SecurityCurrent object is stateful. The methods of the SecurityCurrent object are intended to return information about the state associated with the current execution context. This includes information specific to both the thread of execution that is used to make the call on the SecurityCurrent object, as well as the capsule (process) to which the thread belongs. Changes in state associated purely with the thread, and not with any broader execution context, will remain until the thread terminates or until more state changes are made. State changes associated with a broader execution context (like a process) persist across multiple invocations of methods in the target object, until it is further modified through methods of the SecurityCurrent object or by other means.

Consequently, thread-specific methods called on the SecurityCurrent object are performed on the state associated with the calling thread. The thread in which the SecurityCurrent object was obtained has no influence on the behavior of these methods.

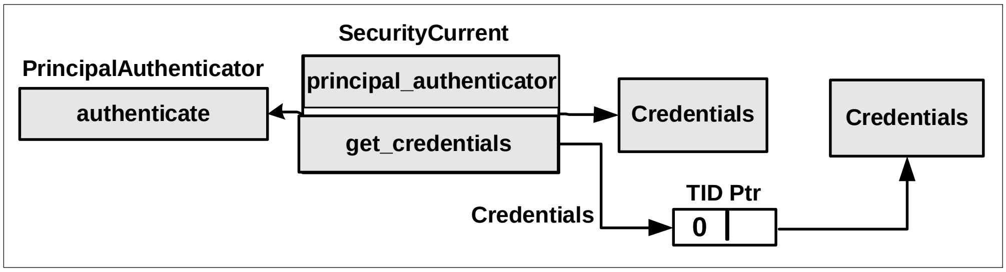
The CORBAservices Security Service specification defines two interfaces for the SecurityCurrent object associated with security:

- `SecurityLevel1.Current`, which derives from `CORBA.Current`
- `SecurityLevel2.Current`, which derives from the `SecurityLevel1.Current` interface

Both interfaces give access to security information associated with the execution context.

At any stage, a client or target object can find the default credentials for subsequent invocations by calling the `Current.get_credentials` method, asking for the invocation credentials. These default credentials are used in all invocations that use object references.

Figure 6-4 SecurityCurrent Object



When the `Current.get_attributes` method is invoked by a client application, the attributes returned from the Credentials object are those of the user.

The SecurityCurrent object is a locality-constrained object. Therefore, a SecurityCurrent object may not be used through the DII/DSI facilities. Any attempt to pass a reference to this object outside of the current process, or any attempt to externalize it using `CORBA.ORB.object_to_string`, results in the raising of the `CORBA.MARSHAL` exception.

Client Security API

The following client security application programming interface (API) modules are implemented as pseudo-objects on the client:

- CORBA module
- TimeBase module
- Security module
- Security Level 1 module
- Security Level 2 module
- Tobj module

The OMG Interface Definition Language (IDL) definitions for these modules are provided in the following sections.

CORBA Module

The Object Management Group (OMG) added the `org.omg.CORBA.Current` interface to the CORBA module to support the SecurityCurrent pseudo-object. The change enables the CORBA module to support Security Replaceability and Security Level 2.

Listing 6-1 shows the `org.omg.CORBA.Current` interface OMG IDL statements.

Listing 6-1 org.omg.CORBA.Current Interface OMG IDL Statements

```
module CORBA {
       // Extensions to CORBA
       interface Current {
       };
};

// This information is taken from CORBAservices: Common Object
// Services Specification, page 15-230. Revised Edition:
```

```
// March 31, 1995. Updated: November 1997. Used with permission by
OMG.
```

TimeBase Module

All data structures pertaining to the basic Time Service, Universal Time Object, and Time Interval Object are defined in the TimeBase module. This allows other services to use these data structures without requiring the interface definitions. The interface definitions and associated enums and exceptions are encapsulated in the TimeBase module.

Listing 6-2 shows the TimeBase module OMG IDL statements.

Listing 6-2 TimeBase Module OMG IDL Statements

```
// From time service
module TimeBase {
       // interim definition of type ulonglong pending the
       // adoption of the type extension by all client ORBs.
       struct ulonglong {
              unsigned long      low;
              unsigned long      high;
       };
       typedef ulonglong         TimeT;
       typedef short             TdfT;
       struct UtcT {
              TimeT              time;     // 8 octets
              unsigned long      inacclo;  // 4 octets
              unsigned short     inacchi;  // 2 octets
              TdfT              tdf;       // 2 octets
                                          // total 16 octets
       };
};

// This information is taken from CORBAservices: Common Object
// Services Specification, p. 14-5. Revised Edition:
// March 31, 1995. Updated: November 1997. Used with permission by
OMG.
```

Table 6-2 defines the TimeBase module data types.

Note: This information is taken from *CORBAservices: Common Object Services Specification*, p. 14-6. Revised Edition: March 31, 1995. Updated: November 1997. Used with permission by OMG.

Table 6-2 TimeBase Module Data Type Definitions

Data Type	Definition
`Time ulonglong`	OMG IDL does not at present have a native type representing an unsigned 64-bit integer. The adoption of technology submitted against that RFP will provide a means for defining a native type representing unsigned 64-bit integers in OMG IDL. Pending the adoption of that technology, you can use this structure to represent unsigned 64-bit integers, understanding that when a native type becomes available, it may not be interoperable with this declaration on all platforms. This definition is for the interim, and is meant to be removed when the native unsigned 64-bit integer type becomes available in OMG IDL.
`Time TimeT`	`TimeT` represents a single time value, which is 64 bit in size, and holds the number of 100 nanoseconds that have passed since the base time. For absolute time, the base is 15 October 1582 00:00.
`Time TdfT`	`TdfT` is of size 16 bits short type and holds the time displacement factor in the form of seconds of displacement from the Greenwich Meridian. Displacements east of the meridian are positive, while those to the west are negative.
`Time UtcT`	`UtcT` defines the structure of the time value that is used universally in the service. When the `UtcT` structure is holding, a relative or absolute time is determined by its history. There is no explicit flag within the object holding that state information. The `inacclo` and `inacchi` fields together hold a value of type `InaccuracyT` packed into 48 bits. The `tdf` field holds time zone information. Implementation must place the time displacement factor for the local time zone in this field whenever it creates a Universal Time Object (UTO). The content of this structure is intended to be opaque; to be able to marshal it correctly, the types of fields need to be identified.

Security Module

The Security module defines the OMG IDL for security data types common to the other security modules. This module depends on the TimeBase module and must be available with any ORB that claims to be security ready.

Listing 6-3 shows the data types supported by the Security module.

Listing 6-3 Security Module OMG IDL Statements

```
module Security {
        typedef sequence<octet>   Opaque;

       // Extensible families for standard data types
       struct ExtensibleFamily {
              unsigned short       family_definer;
              unsigned short       family;
        };

       //security attributes
       typedef unsigned long        SecurityAttributeType;

       // identity attributes; family = 0
       const SecurityAttributeType  AuditId = 1;
       const SecurityAttributeType  AccountingId = 2;
       const SecurityAttributeType  NonRepudiationId = 3;

       // privilege attributes; family = 1
       const SecurityAttributeType  Public = 1;
       const SecurityAttributeType  AccessId = 2;
       const SecurityAttributeType  PrimaryGroupId = 3;
       const SecurityAttributeType  GroupId = 4;
       const SecurityAttributeType  Role = 5;
       const SecurityAttributeType  AttributeSet = 6;
       const SecurityAttributeType  Clearance = 7;
       const SecurityAttributeType  Capability = 8;

       struct AttributeType {
               ExtensibleFamily       attribute_family;
               SecurityAttributeType  attribute_type;
       };

       typedef sequence <AttributeType>   AttributeTypeLists;
       struct SecAttribute {
              AttributeType   attribute_type;
```

```
                Opaque          defining_authority;
                Opaque          value;
                // The value of this attribute can be
               // interpreted only with knowledge of type
        };

        typedef sequence<SecAttribute>  AttributeList;

        // Authentication return status
        enum AuthenticationStatus {
                SecAuthSuccess,
                SecAuthFailure,
                SecAuthContinue,
                SecAuthExpired
        };

        // Authentication method
         typedef unsigned long   AuthenticationMethod;

         enum CredentialType {
                SecInvocationCredentials;
                SecOwnCredentials;
                SecNRCredentials

        // Pick up from TimeBase
        typedef TimeBase::UtcT   UtcT;
};
```

// This information is taken from *CORBAservices: Common Object // Services Specification*, pp. 15-193 to195. Revised Edition: // March 31, 1995. Updated: November 1997. Used with permission by OMG.

Table 6-3 describes the Security module data type.

Table 6-3 Security Module Data Type Definition

Data Type	Definition
`sequence<octet>`	Data whose representation is known only to the Security Service implementation.

Security Level 1 Module

This section defines those interfaces available to client application objects that use only Level 1 Security functionality. This module depends on the CORBA module and the Security and TimeBase modules. The SecurityCurrent interface is implemented by the ORB.

Listing 6-4 shows the Security Level 1 module OMG IDL statements.

Listing 6-4 Security Level 1 Module OMG IDL Statements

```
module SecurityLevel1 {
       interface Current : CORBA::Current {// PIDL
             Security::AttributeList get_attributes(
               in Security::AttributeTypeList  attributes
            );
       };
};

// This information is taken from CORBAservices: Common Object
// Services Specification, p. 15-198. Revised Edition:
// March 31, 1995. Updated: November 1997. Used with permission by
OMG.
```

Security Level 2 Module

This section defines the additional interfaces available to client application objects that use Level 2 Security functionality. This module depends on the CORBA and Security modules.

Listing 6-5 shows the Security Level 2 module OMG IDL statements.

Listing 6-5 Security Level 2 Module OMG IDL Statements

```
module SecurityLevel2 {
       // Forward declaration of interfaces
       interface PrincipalAuthenticator;
```

```
 interface Credentials;
 interface Current;

// Interface Principal Authenticator
 interface PrincipalAuthenticator {
      Security::AuthenticationStatus authenticate(
             in Security::AuthenticationMethod  method,
             in string                         security_name,
             in Security::Opaque               auth_data,
             in Security::AttributeList        privileges,
             out Credentials                   creds,
             out Security::Opaque              continuation_data,
             out Security::Opaque              auth_specific_data
       );

       Security::AuthenticationStatus
                    continue_authentication(
               in Security::Opaque            response_data,
               inout Credentials              creds,
               out Security::Opaque           continuation_data,
               out Security::Opaque           auth_specific_data
       );
  };

 // Interface Credentials
 interface Credentials {
        Security::AttributeList get_attributes(
             in Security::AttributeTypeList   attributes
        );
        boolean is_valid(
               out Security::UtcT      expiry_time
        );
 };

 // Interface Current derived from SecurityLevel1::Current
 // providing additional operations on Current at this
 // security level. This is implemented by the ORB.
 interface Current : SecurityLevel1::Current { // PIDL
        void set_credentials(
               in Security::CredentialType   cred_type,
               in Credentials                cred
        );

        Credentials get_credentials(
             in Security::CredentialType   cred_type
        );
        readonly attribute PrincipalAuthenticator
                  principal_authenticator;
      };
};
```

```
// This information is taken from CORBAservices: Common Object
// Services Specification, pp. 15-198 to 200. Revised Edition:
// March 31, 1995. Updated: November 1997. Used with permission by
// OMG.
```

Tobj Module

This section defines the Tobj module interfaces.

This module provides the interfaces you use to program the BEA TUXEDO style of authentication.

Listing 6-6 shows the Tobj module OMG IDL statements.

Listing 6-6 Tobj Module OMG IDL Statements

```
//Tobj Specific definitions

module Tobj {
       const Security::AuthenticationMethod   TuxedoSecurity =
                                                     0x54555800;

        //get_auth_type () return values
        enum AuthType {
              TOBJ_NOAUTH,
              TOBJ_SYSAUTH,
              TOBJ_APPAUTH
       };

       typedef sequence<octet>    UserAuthData;

      interface PrincipalAuthenticator :
              SecurityLevel2::PrincipalAuthenticator { // PIDL
               AuthType get_auth_type();

              Security::AuthenticationStatus logon(
                     in string           user_name,
                     in string           client_name,
                     in string           system_password,
                     in string           user_password,
                        in UserAuthData      user_data
                  );
                  void logoff();

                 void build_auth_data(
                        in string                     user_name,
                        in string                     client_name,
                        in string                     system_password,
                        in string                     user_password,
                        in UserAuthData               user_data,
                        out Security::Opaque          auth_data,
                        out Security::AttributeList  privileges
                 );
           };
};
```

Java Programming Examples

This section provides programming examples that use the Security Service.

Note: In Listing 6-7, notice that the `resolve_initial_references("SecurityCurrent")` method is used to get a reference to the SecurityCurrent object. The reference is then narrowed, assigned to `cur`, and used to get PrincipalAuthenticator. Refer to Chapter 4, "Bootstrap Object," for a description of this method.

Using WebLogic Enterprise Extensions to Log on

Listing 6-7 shows how to program a Netscape Communicator client using the WebLogic Enterprise extensions to CORBA security. The WebLogic Enterprise extensions enable you to use BEA TUXEDO style authentication. The code in **boldface** shows the OMG method for logging on, which is an alternative to the BEA TUXEDO method. You may prefer the OMG method for log on. Note that the `build_auth_data` method is a BEA-specific method used to prepare data for the OMG method.

Listing 6-7 Java Client Application Using WebLogic Enterprise Extensions to CORBA Security to Log on

```
/*      Copyright (c) 1998 BEA Systems, Inc.
        All rights reserved
        THIS IS PROPRIETARY
        SOURCE CODE OF BEA Systems, Inc.
        The copyright notice above does not
        evidence any actual or intended
        publication of such source code.
*/

//****************************************************************************
//File:SECURITY_CLIENT_EXAMPLE.JAVA
//Description:JAVA  Client program.
//***************************************************************************
```

```
import org.omg.CORBA.*;
import com.beasys.Tobj.*;
import com.beasys.*;
import com.beasys.TobjInternal.*;
import java.io.*;

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

              try {

                 // Initialize ORB
                 ORB orb = ORB.init();

                 // Create Bootstrap Object
                 Tobj_Bootstrap bs = new Tobj_Bootstrap(orb, "");

                 // Get the Security Current Object
                 org.omg.CORBA.Object secCurObj = bs.resolve_initial_references(
                                                         "SecurityCurrent");

                 org.omg.SecurityLevel2.Current secCur =
                       org.omg.SecurityLevel2.CurrentHelper.narrow(secCurObj);

                 // Get a principalauthenticator
                 org.omg.SecurityLevel2.PrincipalAuthenticator authlev2=
                        secCur.principal_authenticator();

                 com.beasys.Tobj.PrincipalAuthenticator auth =
           org.omg.SecurityLevel2.PrincipalAuthenticatorHelper.narrow(authlev2);

                 // Get the auth type
                 com.beasys.Tobj.AuthType authType = auth.get_auth_type();
                 System.out.println( "authType =" + authType);
                 byte[] userData = new byte[0];
                 String userName = "guest";
                 String clientName = "simpclt";
                 String systemPassword = null;
                 String userPassword = null;

                 // Set args according to security level
                 switch (authType.value())
                 {
                         case com.beasys.Tobj.AuthType._TOBJ_NOAUTH:
                                System.out.println(" No Password Required ");
                                break;
                         case com.beasys.Tobj.AuthType._TOBJ_SYSAUTH:
                                System.out.println("System Password Required ");
```

```
                              systemPassword = "security";
                              break;
                       case com.beasys.Tobj.AuthType._TOBJ_APPAUTH:
                            System.out.println("System Password Required & ");
                              System.out.println("User Password Required ");
                              systemPassword = "security";
                              userPassword = "hello";
                              break;
               }
               // Perform Tuxedo style logon
               org.omg.Security.AuthenticationStatus status =
               auth.logon(userName, clientName, systemPassword,
                            userPassword, userData);

               // Prepare args CORBA Seciop style for authentication
                  com.beasys.Tobj.UserAuthData  userData;
                  org.omg.Security.Opaque_var authData;
                  org.omg.Security.AttributeList_var privileges;
                  org.omg.SecurityLevel2.Credentials_var creds;
                  org.omg.Security.Opaque_var continueData;
                  org.omg.CORBA.ULong method = com.beasys.Tobj.TuxedoSecurity;
                  org.omg.Security.Opaque_var authSpecificData;

                  // Use helper to build the authentication data
                  BeaPa->build_auth_data(userName, clientName,
                                         systemPassword,
                                         userPassword,
                                         userData, authData, privileges);
                  // Perform Corba Seciop authentication
                  Security.AuthenticationStatus Status =
                  BeaPa->authenticate(method, userName, authData,
                                         privileges, creds, continueData,
                                         authSpecificData);

              System.out.println( "logon status =" + status);
             if (status != org.omg.Security.AuthenticationStatus.SecAuthSuccess)
                     System.exit(1);
              }
              catch (UserException e){
                      System.err.println("User exception: " + e);
                      e.printStackTrace();
                      System.exit(1);
}
              catch (SystemException e){
                      System.err.println("System exception: " + e);
                      e.printStackTrace();
                      System.exit(1);
              }
```

```
        }
}
```

Getting Information from Privileges

Listing 6-8 shows how to use the Security Service to get information from privileges on a Java client.

Listing 6-8 Getting Information from Privileges

```
try {

      // Build empty attribute list to return all privileges
      org.omg.Security.AttributeType[] type_list =
                     new org.omg.Security.AttributeType[0];
      // Get attributes from current
      org.omg.Security.SecAttribute[] privs =
                     secCur.get_attributes(type_list);
      // Print attributes contents
      for (int i = 0 ; i < privs.length ; i++){
             switch( privs[i].attribute_type.attribute_type){
             case org.omg.Security.Public.value:
                     // No security was specified.
                     // Nothing to print.
                     continue;
             case org.omg.Security.AccessId.value:
                     // User name
                     String user = new String(privs[i].value);
                     System.out.println("User = " + user);
                     continue;
             case org.omg.Security.PrimaryGroupId.value:
                     // Client name
                     String client = new String(privs[i].value);
                     System.out.println("Client = " + client);
                     continue;
             }
      }
}
catch (SystemException e){
       System.out.println("Exception while checking attributes");
       System.exit(1);
}
```

Checking the Validity of the Credentials Expiration Time

Listing 6-9 shows how to use the Security Service to check the validity of the Credentials expiration time on a Netscape Communicator client.

Listing 6-9 Checking Validity of Credentials Expiration Time on a Java Client

```
try {

       // Get Credentials from current
        org.omg.SecurityLevel2.Credentials cred = secCur.get_credentials(
              org.omg.Security.CredentialType.SecInvocationCredentials);
       // Verify credentials
        org.omg.TimeBase.UtcTHolder expiry_time =
              new org.omg.TimeBase.UtcTHolder();
        if (!cred.is_valid(expiry_time)){
              System.out.println(
                   "Credentials are not valid any more");
              System.exit(1);
        }
       // expiry_time contains credentials expiration in
       // 100 nanoseconds since 15 October 1582 00:00
}
catch (SystemException e){
       System.out.println("Exception while checking credentials");
       System.exit(1);
}
```

Authentication Using SecurityLevel2.PrincipalAuthenticator

The following code fragment illustrates the use of the CORBA-compliant interfaces to perform authentication.

```
import org.omg.CORBA.*;
import com.beasys.Tobj.*;
import com.beasys.*;
import com.beasys.TobjInternal.*
import java.io.*;

public class security_client
  {
  public static void main(String[] args)
    {
    Tobj.PrincipalAuthenticator auth = null;

    try
      {
      String  HostPort = args[0];

      // Initialize ORB
      ORB orb = ORB.init();

      // Create bootstrap object
      Tobj_Bootstrap bs =
         new Tobj_Bootstrap(orb, "//” + HostPort);

      // Get security current
      org.omg.CORBA.Object secCurObj =
         bs.resolve_initial_references( "SecurityCurrent" );
      org.omg.SecurityLevel2.Current secCur2Obj =
         org.omg.SecurityLevel2.CurrentHelper.narrow(secCurObj);

      // Get Principal Authenticator
      org.omg.Security.PrincipalAuthenticator princAuth =
         secCur2Obj.principal_authenticator();
      com.beasys.Tobj.PrincipalAuthenticator auth =
         Tobj.PrincipalAuthenticatorHelper.narrow(princAuth);

      // Get Authentication type
      com.beasys.Tobj.AuthType authType = auth.get_auth_type();

      // Initialize arguments
      String userName = "John";
      String clientName = "Teller";
      String systemPassword = null;
      String userPassword = null;
      byte[] userData = new byte[0];

      // Prepare arguments according to security level requested
      switch(authType.value())
        {
        case com.beasys.Tobj.AuthType._TPNOAUTH:
           break;
    case com.beasys.Tobj.AuthType._TPSYSAUTH:
      systemPassword = "sys_pw";
      break;

    case com.beasys.Tobj.AuthType._TPAPPAUTH:
      systemPassword = "sys_pw";
      userPassword = "john_pw";
      break;
    }

  // Build security data
  org.omg.Security.OpaqueHolder auth_data =
     new org.omg.Security.OpaqueHolder();
  org.omg.Security.AttributeListHolder privs =
     new Security.AttributeListHolder();
  auth.build_auth_data(userNname, clientName, systemPassword,
                       userPassword, userData, authData,
                       privs);

  // Authenticate user
  org.omg.SecurityLevel2.CredentialsHolder creds =
     new org.omg.SecurityLevel2.CredentialHolder();
  org.omg.Security.OpaqueHolder cont_data =
     new org.omg.Security.OpaqueHolder();
  org.omg.Security.OpaqueHolder auth_spec_data =
     new org.omg.Security.OpaqueHolder();

  org.omg.Security.AuthenticationStatus status =
     auth.authenticate(com.beasys.Tobj.TuxedoSecurity.value,
                       0, userName, auth_data.value(),
                       privs.value(), creds, cont_data,
                       auth_spec_data);
  if (status != AuthenticatoinStatus.SecAuthSuccess)
    System.exit(1);
  }

catch(UserException e )
  {
  System.err.println( “User exception: “ + e );
  e.printStackTrace();
  System.exit(1);
  }

catch(SystemException e )
  {
  System.err.println( “User exception: “ + e );
  e.printStackTrace();
  System.exit(1);
```

```
          }
        }
      }
```

Authentication Using Tobj.PrincipalAuthenticator

The following code fragment illustrates the use of the BEA extensions to perform authentication.

```
import org.omg.CORBA.*;
import com.beasys.Tobj.*;
import com.beasys.*;
import com.beasys.TobjInternal.*
import java.io.*;

public class security_client
  {
  public static void main(String[] args)
    {
    Tobj.PrincipalAuthenticator auth = null;

    try
      {
      String  HostPort = args[0];

      // Initialize ORB
      ORB orb = ORB.init();

      // Create bootstrap object
      Tobj_Bootstrap bs =
          new Tobj_Bootstrap(orb, "//” + HostPort);

      // Get security current
      org.omg.CORBA.Object secCurObj =
         bs.resolve_initial_references( "SecurityCurrent" );
      org.omg.SecurityLevel2.Current secCur2Obj =
         org.omg.SecurityLevel2.CurrentHelper.narrow(secCurObj);

      // Get Principal Authenticator
      org.omg.Security.PrincipalAuthenticator princAuth =
         secCur2Obj.principal_authenticator();
      com.beasys.Tobj.PrincipalAuthenticator auth =
         Tobj.PrincipalAuthenticatorHelper.narrow(princAuth);

      // Get Authentication type
      com.beasys.Tobj.AuthType authType = auth.get_auth_type();
```

```
        // Initialize arguments
        String userName = "John";
        String clientName = "Teller";
        String systemPassword = null;
        String userPassword = null;
        byte[] userData = new byte[0];

        // Prepare arguments according to security level requested
        switch(authType.value())
          {
          case com.beasys.Tobj.AuthType._TPNOAUTH:
             break;

          case com.beasys.Tobj.AuthType._TPSYSAUTH:
            systemPassword = "sys_pw";
            break;

          case com.beasys.Tobj.AuthType._TPAPPAUTH:
            systemPassword = "sys_pw";
            userPassword = "john_pw";
            break;
          }

        // TUXEDO-style Authenticatation
        org.omg.Security.AuthenticationStatus status =
           auth.logon(userName, clientName, systemPassword,
                      userPassword, userData);

        // Check authentication result
        if (status!= Security.AuthenticationStatus._SecAuthSuccess)
          System.exit(1);
        }

      catch(UserException e )
        {
        System.err.println( “User exception: “ + e );
        e.printStackTrace();
        System.exit(1);
        }

      catch(SystemException e )
        {
        System.err.println( “User exception: “ + e );
        e.printStackTrace();
        System.exit(1);
        }
      // Can now proceed with application
    }
```

Logging Off Using Tobj.PrincipalAuthenticator

The following code fragment illustrates the use of the BEA extensions to log off of a domain.

```
// Log off
    try
      {
      auth.logoff();
      }
    catch (SystemException e)
      {
      }
    }
```

Checking the Validity of Credentials

The following code fragment illustrates the use of the CORBA-compliant interfaces to check the validity of a principal’s credentials.

```
try
  {
  org.omg.Security.UtcTHolder expiry_time =
     new org.omg.Security.UtcTHolder();

// Verify credentials
  if (!cred.is_valid(expiry_time))
    {
    System.out.println( "Credentials are not valid any more" );
    System.exit(1);
    }

  // expiry_time contains credentials expiration in
  // 100 nanoseconds since 15 October 1582 00:00
  }

catch (SystemException e)
  {
  System.out.println( "Exception while checking credentials" );
  System.exit(1);
  }
```

Getting Principal’s Privileges

The following code fragment illustrates the use of the CORBA-compliant interfaces to retrieve the privileges and other attributes from a principal’s credentials.

```
try
  {
  // Build empty attribute type list to return all privileges
  org.omg.Security.AttributeType[] type_list =
     new org.omg.Security.AttributeType[0];

  // Get attributes from current
  org.omg.Security.SecAttribute[] privs =
     creds.get_attributes(type_list);

  // Print attributes contents
  for (int i = 0 ; i < privs.length ; i++)
    {
    switch(privs[i].attribute_type)
      {
      case org.omg.Security.Public.value:
        // No security was specified – Nothing to print.
        continue;
      case org.omg.Security.AccessId.value:
        // User name
        String user = new String(privs[i].value);
        System.out.println( "User = " + user);
        continue;
      case org.omg.Security.PrimaryGroupId.value:
        // Client name
        String client = new String(privs[i].value);
        System.out.println( "Client = " + client);
        continue;
      }
    }
  }
catch (SystemException e)
  {
  System.out.println( "Exception while getting privileges" );
  System.exit(1);
  }
```

Copying a Credentials Object

The following code fragment illustrates the use of the CORBA-compliant interfaces to copy a Credentials object. Copying a Credentials object results in a “deep copy,” possibly creating another security association based on the security technology used by the security provided. Copying a Credentials object that is on the SecurityCurrent object’s “own” list does not place the newly create copy on the “own” list. As a result, the newly created copy of the Credentials object can only be used as the default for one or more threads of the application, and will never be used as a default Credentials object for the capsule (process).

```
try
  {
  org.omg.SecurityLevel2.Credentials   creds_copy =
                secCur2.copy();
  }
catch
  {
  System.out.println( "Exception while copying credential" );
  System.exit(1);
  };
```

Destroying a Credentials Object

The following code fragment illustrates the use of the CORBA-compliant interfaces to destroy a Credentials object. Typically, a Credentials object exists on the “own” list of the SecurityCurrent object. As a result, it should be removed from the “own” list prior to being destroyed. Destroying a Credentials object always results in the destruction of the security association between the client application and the target object, unless the security association is shared with another Credentials object.

```
try
  {
  secCur2.remove_own_credentials( creds );
  secCur2.destroy();
  }
catch
  {
  System.out.println( "Exception while destroying credential" );
  System.exit(1);
  };
```

Getting the Principal Authenticator Object

The following code fragment illustrates the use of the CORBA-compliant interfaces to retrieve the Principal Authenticator object.

```
try
  {
  org.omg.SecurityLevel2.PrincipalAuthenticator princAuth =
     secCurLev2.principal_authenticator();
  }
catch (SystemException e )
  {
  System.err.println( "Error getting principal authenticator" );
  System.exit(1);
  }
```

Getting Credentials

The following code fragment illustrates the use of the CORBA-compliant interfaces to retrieve the privileges and other attributes from a principal's credentials.

```
try
  {
  org.omg.SecurityLevel2.Credentials  creds =
     secCur.get_credentials(
       org.omg.Security.CredentialType.SecInvocatonCredentials);
  }
catch (SystemException e)
  {
  System.out.println( "Exception while getting credentials" );
  System.exit(1);
  }
```

Setting Default Credentials

The following code fragment illustrates the use of the CORBA-compliant interfaces to set the privileges and other attributes for a principal's credentials as the credentials to be used for invocations in the current thread.

```
try
  {
  secCur.set_credentials(
     org.omg.Security.CredentialType.SecInvocationCredentials,
     creds );
  }
catch (SystemException e )
  {
  System.out.println( "Exception while setting credentials" );
  System.exit(1);
  }
```

Getting a Principal's Privileges

The following code fragment illustrates the use of the CORBA-compliant interfaces to retrieve the privileges and other attributes from a principal's credentials.

```
try
  {
  // Build empty attribute type list to return all privileges
  org.omg.Security.AttributeType[] type_list =
     new org.omg.Security.AttributeType[0];

  // Get attributes from current
  org.omg.Security.SecAttribute[] privs =
     secCur.get_attributes(type_list);

  // Print attributes contents
  for (int i = 0 ; i < privs.length ; i++)
    {
    switch(privs[i].attribute_type)
      {
      case org.omg.Security.Public.value:
        // No security was specified – Nothing to print.
        continue;
      case org.omg.Security.AccessId.value:
        // User name
        String user = new String(privs[i].value);
        System.out.println( "User = " + user);
        continue;
      case org.omg.Security.PrimaryGroupId.value:
        // Client name
        String client = new String(privs[i].value);
        System.out.println( "Client = " + client);
```

```
          continue;
        }
      }
    }
  catch (SystemException e)
    {
    System.out.println( "Exception while getting privileges" );
    System.exit(1);
    }
```

Removing a Credentials Object from the “Own” List

The following code fragment illustrates the use of the CORBA-compliant interfaces to remove a Credentials object from the list of default Credentials objects for the current capsule (process). Removing a Credentials object from this list eliminates the ability for the removed Credentials object to be used as the capsule default. It does not destroy the Credentials object, or the security association that it represents.

```
try
  {
  secCur2.remove_own_credentials( creds );
  }
catch
  {
  System.out.println( "Exception while removing credential" );
  System.exit(1);
  };
```

Getting Credentials of the Requesting Principal

The following code fragment illustrates the use of the CORBA-compliant interfaces to retrieve the credentials for the requesting principal.

```
try
  {
  org.omg.SecurityLevel2.ReceivedCredentials recCreds =
     secCurLev2.recevied_credentials();
  org.omg.SecurityLevel2.Credentials creds =
     recCreds.accepting_credentials();
  }
catch (SystemException e )
```

```
  {
  System.err.println( “Exception getting received credentials” );
  System.exit(1);
  }
```

Getting the Principal’s Privileges from Credentials

The following code fragment illustrates the use of the CORBA-compliant interfaces to retrieve the privileges and other attributes from the requesting principal’s credentials.

```
try
  {
  // Build empty attribute list to return all privileges
  org.omg.Security.AttributeType[] type_list =
     new org.omg.Security.AttributeType[0];

  // Get attributes from Credentials
  org.omg.Security.SecAttribute[] privs =
     creds.get_attributes(type_list);

  // Print attributes contents
  for (int i = 0 ; i < privs.length ; i++)
    {
    switch(privs[i].attribute_type)
      {
      case org.omg.Security.Public.value:
        // No security was specified – Nothing to print.
        continue;
      case org.omg.Security.AccessId.value:
        // User name
        String user = new String(privs[i].value);
        System.out.println( "User = " + user);
        continue;
      case org.omg.Security.PrimaryGroupId.value:
        // Client name
        String client = new String(privs[i].value);
        System.out.println( "Client = " + client);
        continue;
      }
    }
  }
catch (SystemException e)
  {
```

```
  System.out.println( "Exception while getting privileges" );
  System.exit(1);
  }
```

Getting the Principal’s Privileges from the SecurityCurrent object

The following code fragment illustrates the use of the CORBA-compliant interfaces to retrieve the privileges and other attributes for the requesting principal from the SecurityCurrent object.

```
try
  {
  // Build empty attribute list to return all privileges
  org.omg.Security.AttributeType[] type_list =
     new org.omg.Security.AttributeType[0];

  // Get attributes from current
  org.omg.Security.SecAttribute[] privs =
     secCurLev2.get_attributes(type_list);

  // Print attributes contents
  for (int i = 0 ; i < privs.length ; i++)
    {
    switch(privs[i].attribute_type)
      {
      case org.omg.Security.Public.value:
        // No security was specified – Nothing to print.
        continue;
      case org.omg.Security.AccessId.value:
        // User name
        String user = new String(privs[i].value);
        System.out.println( "User = " + user);
        continue;
      case org.omg.Security.PrimaryGroupId.value:
        // Client name
        String client = new String(privs[i].value);
        System.out.println( "Client = " + client);
        continue;
      }
    }
  }
catch (SystemException e)
```

```
  {
  System.out.println( "Exception while getting privileges" );
  System.exit(1);
  }
```

Obtaining the SecurityCurrent Object

The following code fragment illustrates how a server application can obtain a reference to the SecurityCurrent object.

```
// Obtain a reference to the bootstrap object
Tobj_Bootstrap bs = TP.bootstrap();
// Get the Security Current
org.omg.CORBA.Object  secCurObj =
  bs.resolve_initial_references( “SecurityCurrent” );
org.omg.SecurityLevel2.Current  secCurLev2
  org.omg.SecurityLevel2.CurrentHelper.narrow(secCurObj);
```

Getting Association Options

The following code fragment illustrates the use of the CORBA-compliant interfaces to get the association options in effect for the secure association with the remote principal.

```
try
  {
  short options = recCreds.association_options_used();
  }
catch (SystemException e)
  {
  System.out.println( "Exception getting association options" );
  System.exit(1);
  }
```

Getting Delegation State

The following code fragment illustrates the use of the CORBA-compliant interfaces to get the delegation state of the remote principal for these credentials.

```
try
  {
  org.omg.Security.DelegationState delState =
         recCreds.delegation_state();
  switch( delState )
    {
    case org.omg.Security.SecInitiator:
      System.out.println( “Acting on own behalf” );
      break;
    case org.omg.Security.SecDelegate:
      System.out.println( “acting on behalf of another” );
      break;
    }
  }
catch (SystemException e)
  {
  System.out.println( "Exception getting delegation state" );
  System.exit(1);
  }
```

Getting Delegation Mode

The following code fragment illustrates the use of the CORBA-compliant interfaces to get the delegation mode of the credentials.

```
try
  {
  org.omg.Security.DelegationMode delMode =
       recCreds.delegation_mode();
  switch( delMode )
    {
    case org.omg.Security.SecDelModeNoDelegation:
      System.out.println( “Unusable for invocation” );
      break;
    case org.omg.Security.SecDelModeSimpleDelegation:
      System.out.println( “Usable for simple delegation” );
      break;
    case org.omg.Security.SecDelModeCompositeDelegation:
      System.out.println( “Usable for composite delegation” );
      break;
    }
  }
catch (SystemException e)
```

```
{
System.out.println( "Exception getting delegation mode" );
System.exit(1);
}
```

CHAPTER

7 Transaction Service

This chapter contains the following topics:

- Capabilities and Limitations. This section describes the following topics:
  - Lightweight Clients with Delegated Commit
  - Transaction Propagation
  - Transaction Integrity
  - Transaction Termination
  - Flat Transactions
  - Interoperability Between Remote Clients and the WebLogic Enterprise Domain
  - Intradomain Interoperability
  - Network Interoperability
  - Relationship of the Transaction Service to Transaction Processing
  - Process Failure
  - Multithreaded Support
  - OMG Interface Definition Language (IDL)
  - General Constraints
- Getting Initial References to the TransactionCurrent Object
- Transaction Service API. This section describes the following topics:
  - Data Types
  - Control Interface
  - TransactionalObject Interface
  - Other CORBAservices Object Transaction Service Interfaces

- Transaction Service API Extensions

The WebLogic Enterprise system provides the following:

- An implementation of the CORBAservices Object Transaction Service (OTS) that is described in Chapter 10 of the *CORBAservices: Common Object Services Specification*. This specification defines the interfaces for an object service that provides transactional functions.

- Sun Microsystems, Inc.’s `javax.transaction` package, which implements the Java Transaction API (JTA).

This chapter describes how the WebLogic Enterprise software implements the OTS; in particular, that portion of the CORBAservices Object Transaction Service that is described as implementation-specific. This chapter provides the information that programmers need to write transactional applications for the WebLogic Enterprise system. It describes the OTS application programming interface (API) that you use to begin or terminate transactions, suspend or resume transactions, and get information about transactions.

For information about JTA, refer to the following:

- The `javax.transaction` package description in the *Java API Reference*.

- The Java Transaction API specification, published by Sun Microsystems, Inc. and available from the Sun Microsystems, Inc. Web site. (See the *Release Notes* for information about obtaining this document.)

Capabilities and Limitations

The following sections describe the capabilities and limitations of the Transaction Service.

Lightweight Clients with Delegated Commit

A lightweight client runs on a single-user, unmanaged desktop system that has irregular availability; that is, the owners may turn their desktop systems off when they are not in use. These single-user, unmanaged desktop systems should not be required to perform network functions like transaction coordination. In particular, unmanaged systems should not be responsible for ensuring atomicity, consistency, isolation, and durability (ACID) properties across failures for transactions involving server resources. WebLogic Enterprise remote clients are lightweight clients.

The Transaction Service allows lightweight clients to do delegated commit. Delegated commit means that the Transaction Service allows lightweight clients to begin and terminate transactions while the responsibility for transaction coordination is delegated to a transaction manager running on a server machine. The lightweight client does not need a local CORBAservices Object Transaction Service transaction manager.

Transaction Propagation

The CORBAservices Object Transaction Service specification states that a client can choose to propagate transaction context either implicitly or explicitly. This implementation of the CORBAservices Object Transaction Service *provides* implicit propagation. Explicit propagation *is strongly discouraged*.

Objects that are related to transaction context that are passed around using explicit transaction propagation *should not* be mixed with implicit transaction propagation APIs. It should be noted, however, that explicit propagation does not place any constraints on when transactional methods can be processed; there is no guarantee that all transactional methods will be completed before the transaction is committed.

Transaction Integrity

Checked transaction behavior provides transaction integrity by guaranteeing that a `commit` will not succeed unless all transactional objects involved in the transaction have completed the processing of their transactional requests. If implicit transaction propagation is used, the Transaction Service *provides* checked transaction behavior that is equivalent to that provided by the request/response interprocess communication models defined by The Open Group. The Transaction Service performs `reply` checks, `commit` checks, and `resume` checks, as described in the *CORBAservices Object Transaction Service Specification.*

Unchecked transaction behavior relies completely on the application to provide transaction integrity. If explicit propagation is used, the Transaction Service *does not* provide checked transaction behavior and transaction integrity *is not* guaranteed.

Transaction Termination

This implementation of the CORBAservices Object Transaction Service allows transactions to be terminated *only* by the client that created the transaction.

Note: The client may be a server object that requests the services of another object.

Flat Transactions

This implementation of the CORBAservices Object Transaction Service implements the flat transaction model.

Interoperability Between Remote Clients and the WebLogic Enterprise Domain

This implementation of the CORBAservices Object Transaction Service *does not* support remote clients invoking methods on server objects in *different* WebLogic Enterprise domains in the *same* transaction.

Remote clients with multiple connections to the same WebLogic Enterprise domain may not make invocations to server objects on these separate connections within the same transaction. An `org.omg.CORBA.NO_PERMISSION` standard system exception is returned to the client.

Intradomain Interoperability

The WebLogic Enterprise implementation of the CORBAservices Object Transaction Service supports native clients invoking methods on server objects in the WebLogic Enterprise domain. In addition, server objects invoking methods on other objects in the same or in different processes in the same WebLogic Enterprise domain is supported.

Network Interoperability

This implementation of the CORBAservices Object Transaction Service does not support the export or import of transactions to or from remote WebLogic Enterprise domains.

Relationship of the Transaction Service to Transaction Processing

This section describes the relationship of the Transaction Service to various transaction processing servers, interfaces, protocols, and standards, as follows:

- Support of BEA TUXEDO ATMI servers

  Servers using the WebLogic Enterprise Transaction Service can make invocations on other BEA TUXEDO Application-to-Transaction Monitor Interface (ATMI) server processes in the same domain. This implementation of the CORBAservices Object Transaction Service *does not* support the following:

  - Remote clients or native clients invoking ATMI services in the WebLogic Enterprise domain
  - ATMI services invoking objects

- Support of The Open Group XA interface

  The Open Group Resource Managers are resource managers that can be involved in a distributed transaction by allowing their two-phase commit protocol to be controlled via The Open Group XA interface. This implementation of the

CORBAservices Object Transaction Service supports interaction with The Open Group Resource Managers.

- Support of the OSI TP protocol

  Open Systems Interconnect Transaction Processing (OSI TP) is the transactional protocol defined by the International Organization for Standardization (ISO). The WebLogic Enterprise implementation of the CORBAservices Object Transaction Service *does not* support interactions with OSI TP transactions.

- Support of the LU 6.2 protocol

  Systems Network Architecture (SNA) LU 6.2 is a transactional protocol defined by IBM. The WebLogic Enterprise implementation of the CORBAservices Object Transaction Service *does not* support interactions with LU 6.2 transactions.

- Support of the ODMG standard

  ODMG-93 is a standard defined by the Object Database Management Group (ODMG) that describes a portable interface to access Object Database Management Systems. The WebLogic Enterprise implementation of the CORBAservices Object Transaction Service *does not* support interactions with ODMG transactions.

Process Failure

The Transaction Service monitors the participants in a transaction for failures and inactivity. One of the features that distinguishes the BEA TUXEDO system from other distributed application environments is the management tools for keeping the application running when failures occur. Because the WebLogic Enterprise implementation of the CORBAservices Object Transaction Service is built upon the existing BEA TUXEDO transaction management system, it inherits the capabilities of the BEA TUXEDO system for keeping applications running.

Multithreaded Support

The WebLogic Enterprise implementation of the CORBAservices Object Transaction Service supports single-threaded implementations *only*. Specifically, a client with an active transaction *cannot* make requests for the same transaction on multiple threads. However, it is possible to have multiple transactions serially active at the same time in a single thread.

OMG Interface Definition Language (IDL)

The CORBAservices Object Transaction Service OMG IDL is described in detail in Chapter 10 of the *CORBAservices: Common Object Services Specification*. The WebLogic Enterprise implementation of the CORBAservices Object Transaction Service supports a *functionally complete* subset of the CORBAservices Object Transaction Service OMG IDL interfaces. For details, see the section “Transaction Service API” on page 7-9.

General Constraints

The following constraints apply:

♦ The WebLogic Enterprise implementation of the CORBAservices Object Transaction Service imposes a limitation on programmers in that a server application object using transactions from the WebLogic Enterprise Transaction Service library *needs* the WebLogic Enterprise TP Framework functionality. A restriction imposed by the WebLogic Enterprise TP Framework is that a client or a server object *cannot* invoke methods on an object that is infected with another transaction. The method invocation issued by the client or the server will return an exception. For further details on the TP Framework, see Chapter 3, “TP Framework.”

♦ A return from the rollback method on the Current object is asynchronous. A consequence of this is that the objects that were infected by the rolled back transaction get their states cleared by the WebLogic Enterprise TP Framework *a little later*. This implies that *no* other client can infect these objects with a different transaction until the WebLogic Enterprise TP Framework clears the states of these objects. This race condition exists for a very short amount of time and is typically not noticeable in a full-fledged application. A simple workaround for this race condition is to try the appropriate operation after a short (typically a 1-second) delay.

- In the WebLogic Enterprise implementation of the CORBAservices Object Transaction Service, clients using other CORBAservices Object Transaction Service implementations *are not* supported.
- In the WebLogic Enterprise implementation, clients may not make oneway method invocations within the context of a transaction to server objects having the `NEVER`, `OPTIONAL`, or `ALWAYS` transaction policies. No error or exception will be returned to the client because it is a oneway method invocation; however, the method on the server object will not be executed. Also, an appropriate error message will be written to the log. Clients may make oneway method invocations within the context of a transaction to server objects having the IGNORE transaction policy. In this case, the method on the server object will be executed, but not in the context of a transaction. For further details on the transaction policies, see Chapter 2, "Server Description File."

Getting Initial References to the TransactionCurrent Object

To access the Transaction Service API and the extension to the Transaction Service API as described later in this chapter, an application needs to issue the following commands.

1. Create a Bootstrap object.
   For details on creating a Bootstrap object, see Chapter 4, "Bootstrap Object."
2. Invoke the `resolve_initial_reference("TransactionCurrent")` method on the Bootstrap object. The invocation returns a standard CORBA object pointer. For a description of this Bootstrap object method, see Chapter 4, "Bootstrap Object."

3. If an application is interested in only the Transaction Service APIs, an `org.omg.CosTransactions.Current.narrow()` should be issued on the object pointer returned from step 2 above. If an application is interested in the Transaction Service APIs with the extensions, a `com.beasys.Tobj.TransactionCurrent.narrow()` should be issued on the object pointer returned from step 2 above.

Transaction Service API

The following sections describe the portions of the CosTransactions modules that are based on CORBA that are implemented in the WebLogic Enterprise software to support the Transaction Service. For further details, refer to Chapter 10 of the *CORBAservices: Common Object Services Specification*.

The definitions and interfaces supported by the Transaction Service in the WebLogic Enterprise software are as follows:

- Data types
- Control interface
- `org.omg.CosTransactions.TransactionalObject` interface

Data Types

Listing 7-1 shows the supported data types.

Listing 7-1 Data Types Supported by the Transaction Service

```
enum Status {

       StatusActive,
       StatusMarkedRollback,
       StatusPrepared,
       StatusCommitted,
       StatusRolledBack,
       StatusUnknown,
        StatusNoTransaction
        StatusPreparing,
        StatusCommitting,
        StatusRollingBack
};

// This information is taken from CORBAservices: Common Object
// Services Specification, p. 10-15. Revised Edition:
// March 31, 1995. Updated: March 1997. Used with permission by OMG.
```

Control Interface

The Control interface allows a program to explicitly manage or propagate a transaction context. An object that supports the Control interface is implicitly associated with one specific transaction.

TransactionalObject Interface

The `org.omg.CosTransactions.TransactionalObject` interface is used by an object to indicate that it is transactional. By supporting this interface, an object indicates that it wants the transaction context associated with the client thread to be propagated on requests to the object. However, this interface is no longer needed. For details on transaction policies that need to be set to infect objects with transactions, see the sections "Server Description File Syntax" on page 2-3 and "TransactionalObject Interface Not Enforced" on page 3-4.

The CosTransactions module defines the TransactionalObject interface (shown in Listing 7-2). The `org.omg.CosTransactions.TransactionalObject` interface defines no methods. It is simply a marker.

Listing 7-2 TransactionalObject Interface

```
interface TransactionalObject {
};

// This information is taken from CORBAservices: Common Object
// Services Specification, p. 10-30. Revised Edition:
```

```
// March 31, 1995. Updated: November 1997. Used with permission by
// OMG.
```

Other CORBAservices Object Transaction Service Interfaces

All other CORBAservices Object Transaction Service interfaces are not supported. Note that the Current interface described earlier is supported only if it has been obtained from the Bootstrap object. The Control interface described earlier is supported only if it has been obtained using the `get_control` and the `suspend` methods on the Current object.

Transaction Service API Extensions

This section describes specific extensions to the COBRAservices Transaction Service API described earlier. The APIs in this section enable an application to open or close an Open Group Resource Manager.

The following APIs help facilitate participation of resource managers in a distributed transaction by allowing their two-phase commit protocol to be controlled via The Open Group XA interface.

The following definitions and interfaces are defined in the `com.beasys.Tobj` module.

Exception

The following exception is supported:

```
exception RMfailed {};
```

A request raises this exception to report that an attempt to open or close a resource manager failed.

TransactionCurrent Interface

This interface supports all the methods of the Current interface in the CosTransactions module as described in the *Java API Reference*. Additionally, this interface supports APIs to open and close the resource manager.

The Tobj module defines the TransactionCurrent interface, as shown in Listing 7-3.

Listing 7-3 TransactionCurrent Interface

```
Interface TransactionCurrent: CosTransactions::Current {
      void open_xa_rm()
             raises(RMfailed);
      void close_xa_rm()
             raises(Rmfailed);
}
```

Table 7-1 describes APIs that are specific to the resource manager. For more information about these APIs, see the *Java API Reference*.

Table 7-1 Resource Manager APIs for the Current Interface

Method	Description
open_xa_rm	This method opens The Open Group Resource Manager to which this process is linked. A `RMfailed` exception is raised if there is a failure while opening the Resource Manager. Any attempts to invoke this method by remote clients or the native clients raises a `NO_IMPLEMENT` standard system exception.
close_xa_rm	This method closes The Open Group Resource Manager to which this process is linked. An `RMfailed` exception is raised if there is a failure while closing the Resource Manager. A `BAD_INV_ORDER` standard system exception is raised if the function was called in an improper context (for example, the caller is in transaction mode). Any attempts by the remote clients or the native clients to invoke this method raises a `NO_IMPLEMENT` standard system exception.

CHAPTER

8 Interface Repository Interfaces

This chapter contains the following topics:

- Structure and Usage
- Building Client Applications
- Getting Initial References to the InterfaceRepository Object
- Interface Repository Interfaces. This section describes the following topics:
    - Supporting Type Definitions
    - IRObject Interface
    - Contained Interface
    - Container Interface
    - IDLType Interface
    - Repository Interface
    - ModuleDef Interface
    - ConstantDef Interface
    - TypedefDef Interface
    - StructDef
    - UnionDef
    - EnumDef
    - AliasDef

- PrimitiveDef
- ExceptionDef
- AttributeDef
- OperationDef
- InterfaceDef

Note: Most of the information in this chapter is taken from Chapter 8 of the *Common Object Request Broker: Architecture and Specification*. Revision 2.2, February 1998. The OMG information has been modified as required to describe the WebLogic Enterprise implementation of the Interface Repository interfaces. Used with permission by OMG.

The WebLogic Enterprise Interface Repository contains the interface descriptions of the CORBA objects that are implemented within the WebLogic Enterprise domain.

The WebLogic Enterprise Interface Repository is based on the CORBA definition of an Interface Repository. It offers a proper subset of the interfaces defined by CORBA; that is, the APIs that are exposed to programmers are implemented as defined by the *Common Object Request Broker: Architecture and Specification* Revision 2.2. However, not all interfaces are supported. In general, the interfaces required to read from the Interface Repository are supported, but the interfaces required to write to the Interface Repository are not. Additionally, not all TypeCode interfaces are supported.

Administration of the Interface Repository is done using tools specific to the WebLogic Enterprise software. These tools allow the system administrator to create an Interface Repository, populate it with definitions specified in Object Management Group Interface Definition Language (OMG IDL), and then delete interfaces. Additionally, an administrator may need to configure the system to include an Interface Repository server. For a description of the Interface Repository administration commands, see Chapter 10, “Java Development and Administration Commands.”

Several abstract interfaces are used as base interfaces for other objects in the Interface Repository. A common set of operations is used to locate objects within the Interface Repository. These operations are defined in the abstract interfaces IRObject, Container, and Contained described in this chapter. All Interface Repository objects inherit from the IRObject interface, which provides an operation for identifying the actual type of the object. Objects that are containers inherit navigation operations from the Container interface. Objects that are contained by other objects inherit navigation operations from the Contained interface. The IDLType interface is inherited by all

Interface Repository objects that represent OMG IDL types, including interfaces, typedefs, and anonymous types. The TypedefDef interface is inherited by all named noninterface types.

The IRObject, Contained, Container, IDLType, and TypedefDef interfaces are not instantiable.

All string data in the Interface Repository are encoded as defined by the ISO 8859-1 character set.

Note: The Write interface is not documented in this chapter because the WebLogic Enterprise software supports only read access to the Interface Repository. Any attempt to use the Write interface to the Interface Repository will raise the exception `org.omg.CORBA.NO_IMPLEMENT`.

Structure and Usage

The Interface Repository consists of two distinct components: the database and the server. The server performs operations on the database.

The Interface Repository database is created and populated using the `idl2ir` administrative command. For a description of this command, see the command "m3idltojava" on page 10-7. From the programmer's point of view, there is no write access to the Interface Repository. None of the write operations defined by CORBA are supported, nor are set operations on non-read-only attributes.

Read access to the Interface Repository database is always through the Interface Repository server; that is, a client reads from the database by invoking methods that are performed by the server. The read operations as defined by the *CORBA Common Object Request Broker: Architecture and Specification,* Revision 2.2, are described in this chapter.

From the Programmer's Point of View

The interface to a server is defined in the OMG IDL file. How the OMG IDL file is accessed depends on the type of client being built. Three types of clients are considered: stub based, Dynamic Invocation Interface (DII), and ActiveX.

Client applications that use stub-style invocations need the OMG IDL file at build time. The programmer can use the OMG IDL file to generate stubs, and so forth. (For more information, see *Creating Client Applications*.) No other access to the Interface Repository is required.

Client applications that use the Dynamic Invocation Interface (DII) need to access the Interface Repository programmatically. The interface to the Interface Repository is defined in this chapter and is discussed in "Building Client Applications" on page 8-5. The exact steps taken to access the Interface Repository depend on whether the client is seeking information about a specific object, or browsing the Interface Repository to find an interface. To obtain information about a specific object, clients use the `org.omg.CORBA.Object._get_interface` method to obtain an InterfaceDef object. (Refer the *Java API Reference* for a description of this method.) Using the InterfaceDef object, the client can get complete information about the interface.

Before a DII client can browse the Interface Repository, it needs to obtain the object reference of the Interface Repository to start the search. DII clients use the Bootstrap object to obtain the object reference. (For a description of this method, see Chapter 4, "Bootstrap Object.") Once the client has the object reference, it can navigate the Interface Repository, starting at the root.

Note: To use the DII, the OMG IDL file must be stored in the Interface Repository.

Client applications that use ActiveX are not aware that they are using the Interface Repository. From the Interface Repository perspective, an ActiveX client is no different than a DII client. ActiveX clients include the Bootstrap object in the Visual Basic code. Like DII clients, ActiveX clients use the Bootstrap object to obtain the Interface Repository object reference. Once the client has the object reference, it can navigate the Interface Repository, starting at the root.

Note: To use an ActiveX client, the OMG IDL file must be stored in the Interface Repository.

Performance Implications

All run-time access to the Interface Repository is via the Interface Repository server. Because there is considerable overhead in making requests of a remote server application, designers need to be aware of this. For example, consider the interaction required to use an object reference to obtain the necessary information to make a DII invocation on the object reference. The steps are as follows:

1. The client application invokes the `_get_interface` operation on the `org.omg.CORBA.Object` to get the InterfaceDef object associated with the object in question. This causes a message to be sent to the ORB that created the object reference.
2. The ORB returns the InterfaceDef object to the client.
3. The client invokes one or more `_is_a` operations on the object to determine what type of interface is supported by the object.
4. After the client has identified the interface, it invokes the `describe_interface` operation on the Interface object to get a full description of the interface (for example, version number, operations, attributes, and parameters). This causes a message to be sent to the Interface Repository, and a reply is returned.
5. The client is now ready to construct a DII request.

Building Client Applications

Java clients that use the Interface Repository need to link in Interface Repository stubs. How this happens is specific to the vendor. If the client application is using the WebLogic Enterprise ORB, the WebLogic Enterprise software provides the stubs in the `org.omg.CORBA` package, which you should include as part of your server application `jar` file. Therefore, programmers do not need to use the Interface Repository OMG IDL file to build the stubs.

If the client application is using a third-party ORB (for example, Orbix) the programmer must use the mechanisms that are provided by that vendor. This might include generating stubs from the OMG IDL file using the IDL compiler supplied by the vendor, simply linking against the stubs provided by the vendor, or some other mechanism.

Some third-party ORBs provide a local Interface Repository capability. In this case, the local Interface Repository is provided by the vendor and is populated with the interface definitions that are needed by that client.

Getting Initial References to the InterfaceRepository Object

You use the Bootstrap object to get an initial reference to the InterfaceRepository object. For a description of the Bootstrap object method, see Chapter 4, “Bootstrap Object.”

Interface Repository Interfaces

Client applications use the interfaces defined by CORBA to access the Interface Repository. This section contains descriptions of each interface that is implemented in the WebLogic Enterprise software.

Supporting Type Definitions

Several types are used throughout the Interface Repository interface definitions.

```
module CORBA {
    typedef string              Identifier;
    typedef string              ScopedName;
    typedef string              RepositoryId;
    enum DefinitionKind {
       dk_none, dk_all,
       dk_Attribute, dk_Constant, dk_Exception, dk_Interface,
       dk_Module, dk_Operation, dk_Typedef,
       dk_Alias, dk_Struct, dk_Union, dk_Enum,
       dk_Primitive, dk_String, dk_Sequence, dk_Array,
       dk_Repository,
       };
};
```

`Identifiers` are the simple names that identify modules, interfaces, constants, typedefs, exceptions, attributes, and operations. They correspond exactly to OMG IDL identifiers. An `Identifier` is not necessarily unique within an entire Interface Repository; it is unique only within a particular Repository, ModuleDef, InterfaceDef, or OperationDef.

A `ScopedName` is a name made up of one or more identifiers separated by two colons (`::`). The identifiers correspond to OMG IDL scoped names. An absolute `ScopedName` is one that begins with two colons and unambiguously identifies a definition in a Repository. An absolute `ScopedName` in a Repository corresponds to a global name in an OMG IDL file. A relative `ScopedName` does not begin with two colons and must be resolved relative to some context.

A `RepositoryId` is an identifier used to uniquely and globally identify a module, interface, constant, typedef, exception, attribute, or operation. Because RepositoryIds are defined as strings, they can be manipulated (for example, copied and compared) using a language binding's string manipulation routines.

A `DefinitionKind` identifies the type of an Interface Repository object.

IRObject Interface

The IRObject interface (shown below) represents the most generic interface from which all other Interface Repository interfaces are derived, even the Repository itself.

```
module CORBA {
     interface IRObject {
            readonly attribute DefinitionKind def_kind;
      };
};
```

The `def_kind` attribute identifies the type of the definition.

Contained Interface

The Contained interface (shown below) is inherited by all Interface Repository interfaces that are contained by other Interface Repository objects. All objects within the Interface Repository, except the root object (Repository) and definitions of anonymous (ArrayDef, StringDef, and SequenceDef), and primitive types are contained by other objects.

```
module CORBA {
    typedef string VersionSpec;

    interface Contained : IRObject {
        readonly attribute RepositoryId    id;
        readonly attribute Identifier      name;
        readonly attribute VersionSpec     version;
    readonly attribute Container       defined_in;
        readonly attribute ScopedName      absolute_name;
        readonly attribute Repository      containing_repository;
        struct Description {
           DefinitionKind              kind;
           any                         value;
         };

        Description describe ();
        };
};
```

An object that is contained by another object has an `id` attribute that identifies it globally, and a `name` attribute that identifies it uniquely within the enclosing Container object. It also has a `version` attribute that distinguishes it from other versioned objects with the same name. The WebLogic Enterprise Interface Repository does not support simultaneous containment or multiple versions of the same named object.

Contained objects also have a `defined_in` attribute that identifies the Container within which they are defined. Objects can be contained either because they are defined within the containing object (for example, an interface is defined within a module) or because they are inherited by the containing object (for example, an operation may be contained by an interface because the interface inherits the operation from another interface). If an object is contained through inheritance, the `defined_in` attribute identifies the InterfaceDef from which the object is inherited.

The `absolute_name` attribute is an absolute `ScopedName` that identifies a Contained object uniquely within its enclosing Repository. If this object's `defined_in` attribute references a Repository, the `absolute_name` is formed by concatenating the string

"::" and this object's name attribute. Otherwise, the absolute_name is formed by concatenating the absolute_name attribute of the object referenced by this object's defined_in attribute, the string "::", and this object's name attribute.

The containing_repository attribute identifies the Repository that is eventually reached by recursively following the object's defined_in attribute.

The describe operation returns a structure containing information about the interface. The description structure associated with each interface is provided below with the interface's definition. The kind of definition described by the structure returned is provided with the returned structure. For example, if the describe operation is invoked on an attribute object, the kind field contains dk_Attribute and the value field contains an any, which contains the AttributeDescription structure.

Container Interface

The Container interface is used to form a containment hierarchy in the Interface Repository. A Container can contain any number of objects derived from the Contained interface. All Containers, except for Repository, are also derived from Contained.

```
module CORBA {
    typedef sequence <Contained> ContainedSeq;

    interface Container : IRObject {
       Contained lookup (in ScopedName search_name);

       ContainedSeq contents (
           in DefinitionKind        limit_type,
           in boolean               exclude_inherited
            );

       ContainedSeq lookup_name (
           in Identifier            search_name,
           in long                  levels_to_search,
           in DefinitionKind        limit_type,
           in boolean               exclude_inherited
            );

        struct Description {
           Contained                 contained_object;
           DefinitionKind            kind;
           any                       value;
```

```
            };

            typedef sequence<Description>  DescriptionSeq;

           DescriptionSeq describe_contents (
              in DefinitionKind         limit_type,
              in boolean                exclude_inherited,
              in long                   max_returned_objs
               );
        };
};
```

The `lookup` operation locates a definition relative to this container, given a scoped name using the OMG IDL rules for name scoping. An absolute scoped name (beginning with "`::`") locates the definition relative to the enclosing Repository. If no object is found, a nil object reference is returned.

The `contents` operation returns the list of objects directly contained by or inherited into the object. The operation is used to navigate through the hierarchy of objects. Starting with the Repository object, a client uses this operation to list all of the objects contained by the Repository, all of the objects contained by the modules within the Repository, all of the interfaces within a specific module, and so on.

`limit_type`

> If `limit_type` is set to `dk_all`, objects of all types are returned. For example, if this is an InterfaceDef, the attribute, operation, and exception objects are all returned. If `limit_type` is set to a specific interface, only objects of that type are returned. For example, only attribute objects are returned if `limit_type` is set to `dk_Attribute`.

`exclude_inherited`

> If set to TRUE, inherited objects (if there are any) are not returned. If set to FALSE, all contained objects (whether contained due to inheritance or because they were defined within the object) are returned.
> The `lookup_name` operation is used to locate an object by name within a particular object or within the objects contained by that object. The `describe_contents` operation combines the `contents` operation and the `describe` operation. For each object returned by the contents operation, the description of the object is returned (that is, the object's `describe` operation is invoked and the results are returned).

`search_name`

> Specifies which name is to be searched for.

`levels_to_search`
: Controls whether the lookup is constrained to the object the operation is invoked on, or whether the lookup should search through objects contained by the object as well. Setting `levels_to_search` to -1 searches the current object and all contained objects. Setting `levels_to_search` to 1 searches only the current object.

`max_returned_objs`
: Limits the number of objects that can be returned in an invocation of the call to the number provided. Setting the parameter to -1 indicates return all contained objects.

IDLType Interface

The IDLType interface (shown below) is an abstract interface inherited by all Interface Repository objects that represent OMG IDL types. It provides access to the TypeCode describing the type, and is used in defining other interfaces wherever definitions of IDL types must be referenced.

```
module CORBA {
       interface IDLType : IRObject {
             readonly attribute TypeCode          type;
       };
};
```

The `type` attribute describes the type defined by an object derived from IDLType.

Repository Interface

Repository (shown below) is an interface that provides global access to the Interface Repository. The Repository object can contain constants, typedefs, exceptions, interfaces, and modules. As it inherits from Container, it can be used to look up any definition (whether globally defined or defined within a module or an interface) either by `name` or by `id`.

```
module CORBA {
       interface Repository : Container {
            Contained lookup_id (in RepositoryId search_id);
            PrimitiveDef get_primitive (in PrimitiveKind kind);
       };
};
```

The `lookup_id` operation is used to look up an object in a Repository, given its `RepositoryId`. If the Repository does not contain a definition for `search_id`, a nil object reference is returned.

The `get_primitive` operation returns a reference to a PrimitiveDef with the specified kind attribute. All PrimitiveDefs are immutable and are owned by the Repository.

ModuleDef Interface

A ModuleDef (shown below) can contain constants, typedefs, exceptions, interfaces, and other module objects.

```
module CORBA {
       interface ModuleDef : Container, Contained {
       };

        struct ModuleDescription {
               Identifier          name;
               RepositoryId        id;
               RepositoryId        defined_in;
               VersionSpec         version;
         };

};
```

The inherited `describe` operation for a ModuleDef object returns a ModuleDescription.

ConstantDef Interface

A ConstantDef object (shown below) defines a named constant.

```
module CORBA {
       interface ConstantDef : Contained {
              readonly attribute TypeCode      type;
              readonly attribute IDLType       type_def;
              readonly attribute any           value;
       };
       struct ConstantDescription {
               Identifier        name;
               RepositoryId      id;
               RepositoryId      defined_in;
               VersionSpec       version;
               TypeCode          type;
               any               value;
       };
};
```

`type`

Specifies the TypeCode describing the type of the constant. The type of a constant must be one of the simple types (long, short, float, char, string, octet, and so on).

`type_def`

Identifies the definition of the type of the constant.

`value`

Contains the value of the constant, not the computation of the value (for example, the fact that it was defined as “1+2”).

The `describe` operation for a ConstantDef object returns a ConstantDescription.

TypedefDef Interface

A TypedefDef (shown below) is an abstract interface used as a base interface for all named nonobject types (structures, unions, enumerations, and aliases). The TypedefDef interface is not inherited by the definition objects for primitive or anonymous types.

```
module CORBA {
      interface TypedefDef : Contained, IDLType {
      };

      struct TypeDescription {
             Identifier               name;
             RepositoryId             id;
             RepositoryId             defined_in;
             VersionSpec              version;
             TypeCode                 type;
       };
};
```

The inherited `describe` operation for interfaces derived from TypedefDef returns a TypeDescription.

StructDef

A StructDef (shown below) represents an OMG IDL structure definition. It contains the members of the struct.

```
module CORBA {
       struct StructMember {
              Identifier       name;
              TypeCode         type;
              IDLType          type_def;
       };
       typedef sequence <StructMember> StructMemberSeq;

       interface StructDef : TypedefDef, Container{
               readonly attribute StructMemberSeq      members;
       };
};
```

The `members` attribute contains a description of each structure member.

The inherited `type` attribute is a `tk_struct` TypeCode describing the structure.

UnionDef

A UnionDef (shown below) represents an OMG IDL union definition. It contains the members of the union.

```
module CORBA {
      struct UnionMember {
             Identifier    name;
             any           label;
             TypeCode      type;
             IDLType       type_def;
      };
      typedef sequence <UnionMember> UnionMemberSeq;

      interface UnionDef : TypedefDef, Container {
          readonly   attribute TypeCode    discriminator_type;
          readonly   attribute IDLType     discriminator_type_def;
          readonly   attribute UnionMemberSeq   members;
      };
};
```

`discriminator_type` and `discriminator_type_def`
    Describe and identify the union's discriminator type.

`members`
    Contains a description of each union member. The label of each UnionMemberDescription is a distinct value of the `discriminator_type`. Adjacent members can have the same name. Members with the same name must also have the same type. A label with type octet and value 0 (zero) indicates the default union member.

The inherited `type` attribute is a `tk_union` TypeCode describing the union.

EnumDef

An EnumDef (shown below) represents an OMG IDL enumeration definition.

```
module CORBA {
      typedef sequence <Identifier> EnumMemberSeq;

      interface EnumDef : TypedefDef {
             readonly attribute EnumMemberSeq         members;
      };
};
```

`members`
    Contains a distinct name for each possible value of the enumeration.

The inherited `type` attribute is a `tk_enum` TypeCode describing the enumeration.

AliasDef

An AliasDef (shown below) represents an OMG IDL typedef that aliases another definition.

```
module CORBA {
      interface AliasDef : TypedefDef {
            readonly attribute IDLType original_type_def;
      };
};
```

`original_type_def`
: Identifies the type being aliased.

The inherited `type` attribute is a `tk_alias` TypeCode describing the alias.

PrimitiveDef

A PrimitiveDef (shown below) represents one of the OMG IDL primitive types. Because primitive types are unnamed, this interface is not derived from `TypedefDef` or `Contained`.

```
module CORBA {
  enum PrimitiveKind {
    pk_null, pk_void, pk_short, pk_long, pk_ushort, pk_ulong,
    pk_float, pk_double, pk_boolean, pk_char, pk_octet,
    pk_any, pk_TypeCode, pk_Principal, pk_string, pk_objref,
    pk_longlong, pk_ulonglong, pk_longdouble, pk_wchar, pk_wstring
  };

  interface PrimitiveDef: IDLType {
      readonly attribute PrimitiveKind        kind;
  };
};
```

`kind`
: Indicates which primitive type the PrimitiveDef represents. There are no PrimitiveDefs with kind `pk_null`. A PrimitiveDef with kind `pk_string` represents an unbounded string. A PrimitiveDef with kind `pk_objref` represents the OMG IDL type Object.

The inherited `type` attribute describes the primitive type.

All PrimitiveDefs are owned by the Repository. References to them are obtained using `Repository::get_primitive`.

ExceptionDef

An ExceptionDef (shown below) represents an exception definition. It can contain structs, unions, and enums.

```
module CORBA {
     interface ExceptionDef : Contained, Container {
          readonly   attribute TypeCode              type;
          readonly   attribute StructMemberSeq       members;
     };
     struct ExceptionDescription {
          Identifier               name;
          RepositoryId             id;
          RepositoryId             defined_in;
          VersionSpec              version;
          TypeCode                 type;
     };
};
```

`type`

`tk_except` TypeCode that describes the exception.

`members`

Describes any exception members.

The `describe` operation for a ExceptionDef object returns an ExceptionDescription.

AttributeDef

An AttributeDef (shown below) represents the information that defines an attribute of an interface.

```
module CORBA {
      enum AttributeMode {ATTR_NORMAL, ATTR_READONLY};
      interface AttributeDef : Contained {
          readonly       attribute TypeCode          type;
                         attribute IDLType           type_def;
                         attribute AttributeMode     mode;
      };
      struct AttributeDescription {
           Identifier         name;
           RepositoryId       id;
           RepositoryId       defined_in;
           VersionSpec        version;
           TypeCode           type;
           AttributeMode      mode;
      };
};
```

`type`
: Provides the TypeCode describing the type of this attribute.

`type_def`
: Identifies the object that defines the type of this attribute.

`mode`
: Specifies read only or read/write access for this attribute.

OperationDef

An OperationDef (shown below) represents the information needed to define an operation of an interface.

```
module CORBA {
      enum OperationMode {OP_NORMAL, OP_ONEWAY};

      enum ParameterMode {PARAM_IN, PARAM_OUT, PARAM_INOUT};
      struct ParameterDescription {
            Identifier            name;
            TypeCode              type;
            IDLType               type_def;
            ParameterMode         mode;
      };
      typedef sequence <ParameterDescription> ParDescriptionSeq;

      typedef Identifier ContextIdentifier;
      typedef sequence <ContextIdentifier> ContextIdSeq;

      typedef sequence <ExceptionDef> ExceptionDefSeq;
      typedef sequence <ExceptionDescription> ExcDescriptionSeq;

      interface OperationDef : Contained {
            readonly    attribute TypeCode              result;
            readonly    attribute IDLType               result_def;
            readonly    attribute ParDescriptionSeq     params;
            readonly    attribute OperationMode         mode;
            readonly    attribute ContextIdSeq          contexts;
            readonly    attribute ExceptionDefSeq       exceptions;
       };

       struct OperationDescription {
           Identifier                   name;
           RepositoryId                 id;
           RepositoryId                 defined_in;
           VersionSpec                  version;
           TypeCode                     result;
           OperationMode                mode;
           ContextIdSeq                 contexts;
           ParDescriptionSeq            parameters;
           ExcDescriptionSeq            exceptions;
      };
};
```

`result`

A TypeCode that describes the type of the value returned by the operation.

`result_def`

Identifies the definition of the returned type.

`params`

Describes the parameters of the operation. It is a sequence of ParameterDescription structures. The order of the ParameterDescriptions in the sequence is significant. The `name` member of each structure provides the parameter name. The `type` member is a TypeCode describing the type of the parameter. The `type_def` member identifies the definition of the type of the parameter. The `mode` member indicates whether the parameter is an in, out, or inout parameter.

`mode`

The operation's `mode` is either oneway (that is, no output is returned) or normal.

`contexts`

Specifies the list of context identifiers that apply to the operation.

`exceptions`

Specifies the list of exception types that can be raised by the operation.

The inherited `describe` operation for an OperationDef object returns an OperationDescription.

The inherited `describe_contents` operation provides a complete description of this operation, including a description of each parameter defined for this operation.

InterfaceDef

An InterfaceDef object (shown below) represents an interface definition. It can contain constants, typedefs, exceptions, operations, and attributes.

```
module CORBA {
   interface InterfaceDef;
      typedef sequence <InterfaceDef> InterfaceDefSeq;
      typedef sequence <RepositoryId> RepositoryIdSeq;
      typedef sequence <OperationDescription> OpDescriptionSeq;
      typedef sequence <AttributeDescription> AttrDescriptionSeq;

      interface InterfaceDef : Container, Contained, IDLType {

           readonly attribute InterfaceDefSeq   base_interfaces;

           boolean is_a (in RepositoryId interface_id);

          struct FullInterfaceDescription {
              Identifier             name;
              RepositoryId           id;
              RepositoryId           defined_in;
              VersionSpec            version;
              OpDescriptionSeq       operations;
              AttrDescriptionSeq     attributes;
              RepositoryIdSeq        base_interfaces;
              TypeCode               type;
          };

           FullInterfaceDescription describe_interface();

     };

     struct InterfaceDescription {
          Identifier                name;
```

```
            RepositoryId                id;
            RepositoryId                defined_in;
            VersionSpec                 version;
            RepositoryIdSeq             base_interfaces;
        };
    };
```

`base_interfaces`

Lists all the interfaces from which this interface inherits. The `is_a` operation returns TRUE if the interface on which it is invoked either is identical to or inherits, directly or indirectly, from the interface identified by its `interface_id` parameter. Otherwise, it returns FALSE.

The `describe_interface` operation returns a FullInterfaceDescription describing the interface, including its operations and attributes.

The inherited `describe` operation for an InterfaceDef returns an InterfaceDescription.

The inherited `contents` operation returns the list of constants, typedefs, and exceptions defined in this InterfaceDef and the list of attributes and operations either defined or inherited in this InterfaceDef. If the `exclude_inherited` parameter is set to TRUE, only attributes and operations defined within this interface are returned. If the `exclude_inherited` parameter is set to FALSE, all attributes and operations are returned.

CHAPTER

9 Joint Client/Server Applications

This chapter contains the following topics:

- Introduction. This section includes the following topics:
  - Main Program and Server Initialization
  - Servants
  - Servant Inheritance from Skeletons
  - Callback Object Models Supported
  - Preparing Callback Objects using BEAWrapper Callbacks
  - Threading Considerations in the Main Program
  - Java Client ORB Initialization
  - IIOP Support
- Callbacks Interface API

This chapter describes programming requirements for joint client/server applications. For a description of the BEAWrapper package and the `Callbacks` interface API, see the *Java API Reference*.

Introduction

For either a WebLogic Enterprise client applications or a joint client/server application (that is, a client that can receive and process object invocations), create a Java client `main()` method. The `main()` method uses WebLogic Enterprise environmental objects to establish connections, set up security, and start transactions.

WebLogic Enterprise clients invoke operations on objects. In the case of DII, client code creates the DII Request object and then invokes one of two operations on the DII Request. In the case of static invocation, client code performs the invocation by performing what looks like an ordinary Java invocation (which ends up calling code in the generated client stub). Additionally, the client programmer uses ORB interfaces defined by OMG and WebLogic Enterprise environmental objects that are supplied with the WebLogic Enterprise software to perform functions unique to WebLogic Enterprise.

For WebLogic Enterprise joint client/server applications, the client code must be structured so that it can act as a server for callback WebLogic Enterprise objects only. Such clients do not use the TP Framework and are not subject to WebLogic Enterprise system administration. Besides the programming implications, this means that joint client/server applications do not have the same scalability and reliability as WebLogic Enterprise servers, nor do they have the state management and transaction behavior available in the TP Framework. If a user wants to have those characteristics, the application must be structured in such a way that the object implementations are in a WebLogic Enterprise server, rather than in a client.

The following sections describe the mechanisms you use to add callback support to a WebLogic Enterprise client. In some cases, the mechanisms are contrasted with the WebLogic Enterprise server mechanisms that use the TP Framework.

Main Program and Server Initialization

In a WebLogic Enterprise Java server, you use the `buildjavaserver` command to create the main program for the server. The server main program takes care of all WebLogic Enterprise- and CORBA-related initialization of the server functions. However, since you implement the Server object, you have an opportunity to customize the way in which the server application is initialized and shut down. The server main program automatically invokes methods on the Server object at the appropriate times.

In contrast, for a WebLogic Enterprise joint client/server application (as for a WebLogic Enterprise client), you create the main program and are responsible for all initialization. You do not need to provide a Server object because you have complete control over the main program and you can provide initialization and shutdown code in any way that is convenient.

The specific initialization needed for a joint client/server application is discussed in the section “Servants” on page 9-3.

Servants

Servants (method code) for WebLogic Enterprise joint client/server applications are very similar to servants for WebLogic Enterprise servers. All business logic is written the same way. The differences result from not using the TP Framework, which includes the `Server`, `TP`, and `Tobj_Servant` classes. Therefore, the main difference is that you use CORBA functions directly instead of indirectly through the TP Framework.

In WebLogic Enterprise Java server applications, servants are created dynamically. However, in WebLogic Enterprise joint client/server applications, the user application is responsible for creating a servant before any requests arrive; thus, the `Server` class is not needed. Typically, the program creates a servant, initializes it, and then activates the object. The process of activation, which associates the servant with an object ID (either user supplied or system generated), results in the creation of an object reference that the server application subsequently can provide to another process. Such an object might be used to handle callbacks. Thus, the servant already exists, and the object is already active, before a request for that object arrives.

Instead of invoking the `TP` interface to perform certain operations, client servants directly invoke the ORB and the BOA (for clients that are based on the Java JDK ORB). Alternately, since much of the interaction with the ORB and the BOA is the same for all applications, the join client/server library (`wleclient.jar`) provides a convenience wrapper object (`Callbacks`) that does the same things using a single operation. In addition, the wrapper objects also provide extra POA-like life span policies for `ObjectIds`, see “Callback Object Models Supported” on page 9-4 and “Preparing Callback Objects using BEAWrapper Callbacks” on page 9-6.

Servant Inheritance from Skeletons

In a WLE client, as well as in a WLE server, a user-written Java implementation class inherits from the same skeleton class name generated by the `idltojava` compiler. For example, given the IDL:

```
interface Hospital{ … };
```

The skeleton generated by `idltojava` contains a skeleton class, `_HospitalImplBase`, from which the user-written class inherits, as in:

```
class HospitalImpl extends _HospitalImplBase {…};
```

In a WLE server application, the skeleton class inherits from the TP Framework class `com.beasys.Tobj_Servant`, which in turn inherits from the CORBA-defined class `org.omg.PortableServer.Servant`.

The inheritance tree for a callback object implementation in a joint client/server application is different from that of a client. The skeleton class does not inherit from the TP Framework class, but instead inherits from the `org.omg.CORBA.DynamicImplementation` class, which in turn inherits from the `org.omg.CORBA.portable.ObjectImpl` class.

Not having the `Tobj_Servant` class in the inheritance tree for a servant means that the servant does not have the `activate_object` and `deactivate_object` methods. In a WLE server application, these methods are invoked by the TP Framework to dynamically initialize and save a servant's state before invoking a method on the servant. For a joint client/server application, user code must explicitly create a servant and initialize a servant's state; therefore, the `Tobj_Servant` operations are not needed.

Callback Object Models Supported

WebLogic Enterprise software supports the three kinds of callback objects. These object types are described here primarily in terms of their behavioral characteristics rather than in the details about how the ORB and the wrapper classes handle them.

The three kinds of callback objects are:

- **Transient/SystemId**

  Object references are valid only for the life of the client process. The `objectId` is not assigned by the client application, but is a unique value assigned by the system. This type of object is useful for invocations that a client wants to receive only until the client terminates. If used with a Notification or Event Service, for example, these are callbacks that correspond to the concept of transient events and transient channels. (The corresponding POA LifeSpanPolicy value is `TRANSIENT`, and the IdAssignmentPolicy is `SYSTEM_ID`.)

- **Persistent/SystemId**

  Object references are valid across multiple activations. The `objectId` is not assigned by the client application, but is a unique value assigned by the system. This type of object and object reference is useful when the client goes up and down over a period of time. When the client is up, it can receive callback objects on that particular object reference. Typically, the client creates the object reference once, saves it in its own permanent storage area, and reactivates the servant for that object every time the client comes up. If used with a Notification Service, for example, these are callbacks that correspond to the concept of a persistent subscription; that is, the Notification Service remembers the callback reference and delivers events any time the client is up and declares that it is again ready to receive them. This allows notification to survive client failures or offline-time. (The corresponding POA policy values are `PERSISTENT` and `SYSTEM_ID`.)

- **Persistent/UserId**

  This is the same as **Persistent/SystemId**, except that the `objectId` has to be assigned by the client application. Such an `objectId` might be, for example, a database key meaningful only to the client. (The corresponding POA policy values are `PERSISTENT` and `USER_ID`.)

Note: The **Transient/UserId** policy combination is not considered particularly important. In any event, this policy combination is not available in Java server applications.

Note: For WebLogic Enterprise native joint client/server applications, neither of the Persistent policies is supported, only the Transient policy.

In C++, these object models are established by using combinations of the following POA policies, which control both the types of objects and the types of object references that are possible:

- LifeSpanPolicy, which controls how long an object reference is valid
- IdAssignmentPolicy, which controls who assigns the `objectId`—the user or the system

However, since the ORB used for Java server applications does not provide a POA, the WLE system provides a `Callbacks` wrapper class that emulates these POA policies.

Preparing Callback Objects using BEAWrapper Callbacks

Because the code to prepare for callback objects is nearly identical for every joint client/server application, and because the Java JDK ORB does not implement a POA, WLE provides a wrapper class in the joint client/server library that is virtually identical to the wrapper class provided in C++. This wrapper class emulates the POA policies needed to support the three types of callback objects.

The following code shows the `Callback` wrapper interfaces.

```
package com.beasys.BEAWrapper;

       class Callbacks{
             public Callbacks ();

             public Callbacks (org.omg.CORBA.Object init_orb);

             public org.omg.CORBA.Object start_transient (
                                org.omg.PortableServer.ObjectImpl servant,
                                java.lang.String rep_id)
                                throws ServantAlreadyActive,
                                       org.omg.CORBA.BAD_PARAMETER;

             public org.omg.CORBA.Object start_persistent_systemid (
                                org.omg.PortableServer.ObjectImpl servant,
                                java.lang.String rep_id,
                                org.omg.CORBA.StringHolder stroid)
                                throws ServantAlreadyActive,
                                       org.omg.CORBA.BAD_PARAMETER,
                                       org.omg.CORBA.IMP_LIMIT;

             public org.omg.CORBA.Object restart_persistent_systemid (
                         org.omg.PortableServer.ObjectImpl servant,
                         java.lang.String rep_id,
                         java.lang.String stroid)
                         throws ServantAlreadyActive,
                                ObjectAlreadyActive,
                                org.omg.CORBA.BAD_PARAMETER,
                                org.omg.CORBA.IMP_LIMIT;

          public org.omg.CORBA.Object start_persistent_userid (
                         org.omg.PortableServer.ObjectImpl servant,
                         java.lang.String rep_id,
                         java.lang.String stroid)
                         throws ServantAlreadyActive,
                                ObjectAlreadyActive,
                                org.omg.CORBA.BAD_PARAMETER,
                                org.omg.CORBA.IMP_LIMIT;

          public void stop_object(
                         org.omg.PortableServer.ObjectImpl
                                                          servant);

          public String get_string_oid ()
                         throws NotInRequest;

          public void stop_all_objects();


};
```

Threading Considerations in the Main Program

When a program acts as both a client and a server in a Java client, those two parts can execute concurrently in different threads. Since Java as an execution environment is inherently multithreaded, there is no reason to invoke the `org.omg.CORBA.orb.work_pending` and `org.omg.CORBA.orb.perform_work` methods from a Java client. In fact, if the Java client tries to invoke these methods, these methods throw an `org.omg.CORBA.NO_IMPLEMENT` exception. The client does not need to invoke the `org.omg.CORBA.orb.run` method. As in any multithreaded environment, any code that may execute concurrently (client and servant code for a callback) in the client application must be coded to be thread safe. This is a departure from C++ clients, which are currently single-threaded.

Multiple Threads

In Java, the client starts up in the main thread. The client can then set up callback objects via an invocation to any of the *(re)*`start_`*xxxx* methods provided by the Callbacks wrapper class. The wrapper class handles registering the servant and its associated OID in the ORB's object manager. The application is then free to pass the object reference returned by the *(re)*`start_`*xxxx* method to an application that needs to call back to the servant.

Note: The ORB requires an explicit invocation to one of the *(re)*`start_`*xxxx* methods to effectively initialize the servant and create a valid object reference that can be marshaled properly to another application. This is a deviation from the base JDK 1.2 ORB behavior that allows implicit object reference creation via an internal invocation to the `orb.connect` method when marshaling an object reference, if the application has not yet done so.

Invocations on the callback object are handled by the ORB. As each request is received, the ORB validates the request against the object manager and spawns a thread for that request. Multiple requests can be made simultaneously to the same object because the ORB creates a new thread for each request; that is why the Servant code of the Callback must be written thread safe. As each request terminates, the thread that runs the servant also terminates.

The main client thread can make as many client invocations as necessary. An invocation to the `stop_`*(all_)*`object` methods merely takes the object out of the object manager's list, thereby preventing any further invocations on it. Any invocation to a stopped object fails as if it never existed.

If the client application needs to retrieve the results of a callback from another thread, the client application must use normal thread synchronization techniques to do so.

If any thread (client main or servant) in the WLE remote-like client application exits, all the client process activity is stopped, and the Java execution environment terminates. We recommend only to invoke the `return` method to terminate a thread.

Java Client ORB Initialization

A client application must initialize the ORB with the BEA-supplied properties. This is so that the ORB will utilize the BEA-supplied classes and methods that support the `Callbacks` wrapper class and the Bootstrap object. You can find these classes in wleclient.jar, which is installed in $TUXDIR/udataobj/java/jdk (on Solaris) or %TUXDIR%\udataobj\java\jdk (on Windows NT). The application must set certain system properties to do this, as shown in the following example:

```
Properties prop = new Properties(System.getProperties());
prop.put("org.omg.CORBA.ORBClass","com.beasys.CORBA.iiop.ORB");
prop.put("org.omg.CORBA.ORBSingletonClass",
         "com.beasys.CORBA.idl.ORBSingleton");
System.setProperties(prop);
// Initialize the ORB.
ORB orb = ORB.init(args, prop);
```

IIOP Support

IIOP is the protocol used for communication between ORBs. IIOP allows ORBs from different vendors to interoperate. For Java server applications, a port number must be supplied at the client for persistent or user ID object reference policies.

Java Applet Support

IIOP support for applets that want to receive callbacks or callouts is limited due to applet security mechanisms. Any applet run-time environment that allows an applet to create and listen on sockets (via their proprietary environment or protocol) will be able to act as WLE joint client/server applications. If the applet run-time environment restricts socket communication, then the applet cannot be a joint client/server application to a WLE application.

Port Numbers for Persistent Object References

WLE Java server applications support only GIOP V1.0, as described in Chapter 13 of the OMG CORBA 2.2 specification.

For a WLE Java remote joint client/server application to support IIOP, the object references created for the server component must contain a host and a port. For transient object references, any port is sufficient and can be obtained by the ORB dynamically; however, this is not sufficient for persistent object references.

Persistent references must be served on the same port after the ORB restarts. That is, the ORB must be prepared to accept requests on the same port with which it created the object reference. Therefore, there must be some way to configure the ORB to use a particular port.

Java clients that expect to act as servers for callbacks of persistent references must now be started with a specified port. This is done by setting the system property `org.omg.CORBA.ORBPort,` as in the following commands:

For Windows NT:

```
java -DTOBJADDR=//host:port
     -Dorg.omg.CORBA.ORBPort=xxxx
     -classpath=%CLASSPATH% client
```

For Unix:

```
java -DTOBJADDR=//host:port
     -Dorg.omg.CORBA.ORBPort=xxxx
     -classpath=$CLASSPATH client
```

Typically, a system administrator assigns the port number for the client from the user range of port numbers, rather from the dynamic range. This keeps the joint client/server applications from using conflicting ports.

If a WLE remote joint client/server application tries to create a persistent object reference without having set a port (as in the preceding command line), the operation raises an exception, `IMP_LIMIT`, informing the user that a truly persistent object reference cannot be created.

Callbacks Interface API

For a complete description of the `BEAWrapper.Callbacks` interface API, see the *Java API Reference*

10 Java Development and Administration Commands

This chapter describes the following commands:

- `buildjavaserver`
- `buildXAJS`
- `m3idltojava`

This chapter is an alphabetical reference that describes each WebLogic Enterprise development command and Interface Repository administration command for developing Java applications for the Windows NT and UNIX environments. A list of valid parameters and options is shown for each command. For information about building C++ client and server applications, see the *C++ Programming Reference*.

Note: For descriptions of the `idl2ir`, `irdel`, and `ir2idl` commands, see the *Administration Guide*.

Before executing a WebLogic Enterprise command, you must ensure that the WebLogic Enterprise `bin` directory is in your defined path:

On Windows NT:

```
Set Path=%TUXDIR%\Bin;%Path%
```

On UNIX:

For c shell (csh): `set path = ($TUXDIR/bin $path)`

For Bourne (sh) or Korn (ksh): `PATH=$TUXDIR/bin:$PATH`

```
                                export PATH
```

Before executing a WebLogic Enterprise command, you must set the environment variables that are listed with each command.

On Windows NT systems, the syntax for setting an environment variable is:

```
set var=value
```

On UNIX systems, the syntax for setting an environment variable is:

- For c shell:

  ```
  setenv var value
  ```

- For Bourne and Korn (sh/ksh):

  ```
  var=value
  export var
  ```

buildjavaserver

Synopsis

Constructs a Java WebLogic Enterprise server application `jar` file.

Syntax

`buildjavaserver [-s` *`searchpath`*`]` *`input_file`*

Description

Once the class files that make up a server application have been created and specified, along with interface activation and transaction policies, in the Server Description File, you use the `buildjavaserver` command to create the `jar` file. The `jar` file contains all the server application class files and a server descriptor. The server descriptor is a serialized Java object that contains:

- ♦ Information about all the servant classes implemented by the server application
- ♦ Activation and transaction policies for all the interfaces that have been defined in the application's OMG IDL file
- ♦ The name of the Server object, which initializes and stops the server application and performs object housekeeping

Options

`-s`

Specifies a path to be used by the `buildjavaserver` command to locate the classes and packages needed for building the `jar` file. If you do not specify this option, the `buildjavaserver` command uses the class path by default.

`input_file`

Specifies the name of the Server Description File. For information about creating this file, see Chapter 2, "Server Description File."

Environment Variables

`TUXDIR`

Finds the WebLogic Enterprise libraries and include files to use when compiling the server application.

`LD_LIBRARY_PATH` (Solaris systems)

Indicates which directories contain shared objects to be used by the compiler, in addition to the WebLogic Enterprise shared objects. A colon (`:`) is used to separate the list of directories.

`LIB` (Windows NT systems)

Indicates a list of directories within which to find libraries. A semicolon (;) is used to separate the list of directories.

Portability

The `buildjavaserver` command is not supported on client-only WebLogic Enterprise systems.

Example

The following example builds a Java WebLogic Enterprise server application `jar` file on a Solaris system. This example uses the `com/acme` path for locating classes and packages for the archive and also uses the Server Description File `MyServer.xml`.

```
buildjavaserver -s com/acme MyServer.xml
```

buildXAJS

Synopsis Constructs an XA resource manager to be used with a Java server application group.

Syntax `buildXAJS [-v] -r *rmname* [-o *outfile*]`

Description Use this command to build an XA resource manager that you want to use with a Java server application group. In the application's `UBBCONFIG` file, you use the `JavaServerXA` element in place of the `JavaServer` element to associate the XA resource manager with a specified server group. Note that a server application configured to use the default XA resource manager (that is, `NULL`) cannot coexist in a server group that uses a nondefault XA resource manager, such as Oracle. Refer to the *Administration Guide* for more information about configuring server groups with an XA resource manager.

Options

`-v`

Specifies that the `buildXAJS` command should work in verbose mode. In particular, it writes the build command to its standard output.

`-r` *`rmname`*

Specifies the resource manager associated with this server. The value `rmname` must appear in the resource manager table located in `$TUXDIR/udataobj/RM` on Solaris systems, or `%TUXDIR%\udataobj\RM` on Windows NT systems. On Solaris systems, each entry in this file is of the form `rmname:rmstructure_name:library_names`. On NT systems, each entry in this file is of the form `rmname;rmstructure_name;library_names`. Using the `rmname` value, the entry in `$TUXDIR/udataobj/RM` or `%TUXDIR%\udataobj\RM` automatically includes the associated libraries for the resource manager and properly sets up the interface between the transaction manager and the resource manager. The value `TUXEDO/SQL` includes the libraries for the BEA TUXEDO System/SQL resource manager. Other values can be specified as they are added to the resource manager table. If the `-r` option is not specified, the default is to use the null resource manager.

`-o` *`outfile`*

Specifies the name of the output file. If no name is specified, the default is `JavaServerXA`.

Environment Variables

`TUXDIR`

Finds the WebLogic Enterprise libraries and include files to use when compiling the server application.

`LD_LIBRARY_PATH` (Solaris systems)
: Indicates which directories contain shared objects to be used by the compiler, in addition to the WebLogic Enterprise shared objects. A colon (:) is used to separate the list of directories.

`LIB` (Windows NT systems)
: Indicates a list of directories within which to find libraries. A semicolon (;) is used to separate the list of directories.

Portability The `buildXAJS` command is not supported on client-only WebLogic Enterprise systems.

Example The following example builds a Java server XA resource manager on a Solaris system:

```
buildXAJS -r oracle7
```

m3idltojava

Synopsis

Compiles the Object Management Group (OMG) Interface Definition Language (IDL) file and generates client stub and server skeleton files required for the interface definitions being implemented in Java. Use this command only when you are creating a Java server application.

Syntax

```
m3idltojava [-p] [-j javaDirectory] [-Idirectory][-Dsymbol]
           [-Usymbol] [-foptions] idl-filename...
```

Description

The `m3idltojava` command compiles OMG IDL source files into Java source code. You then use the javac compiler to compile that source into Java bytecodes. The OMG IDL declarations from the named OMG IDL files are translated to Java declarations according to the mapping from OMG IDL to Java.

Given the provided *`idl-filename`* file(s), the `m3idltojava` command generates the following files for each interface defined in the server application's OMG IDL file:

`interface-name``.java`
: Contains the Java version of the interface definitions in the OMG IDL file. Each interface implementation extends the `org.omg.CORBA.Object` class.

`_interface-name``Stub.java`
: Is the client stub file.

`_interface-name``ImplBase.java`
: Is the Server skeleton file, which is extended by the server application's object implementation classes.

`interface-name``Helper.java`
: Contains the helper class for the object.

`interface-name``Holder.java`
: Contains the holder class for the object.

The `m3idltojava` compiler generates the client stub and server skeleton files. Any previous versions are overwritten.

If an unknown option is passed to this command, the offending option and a usage message is displayed to the user, and the compile is not performed.

For more information about OMG IDL syntax, see Chapter 1, "OMG IDL Syntax."

Parameter

`idl-filename`
: Represents the name of one or more files that contain OMG IDL statements.

Options

`-p` *`package`*

Specifies that generated Java classes should be part of the given package. The compiler creates the appropriate directory hierarchy and stores the generated files in the directory that corresponds to their package. If you specify the `-j` option, the hierarchy is created under the specified directory. Otherwise, the hierarchy is created under the current directory. You can override this option by using `#pragma javaPackage` in the OMG IDL source file.

`-j` *`javaDirectory`*

Specifies that generated Java files should be written to the specified directory. This directory is independent of the `-p` option, if used.

`-I`*`directory`*

Specifies directories within which to search for include files, in addition to any directories specified with the `#include` OMG IDL preprocessor directive. Multiple directories can be specified by using multiple `-I` options.

There are two types of `#include` OMG IDL preprocessor directives: `system` (for example, `<a.idl>`) and `user` (for example, `"a.idl"`). The path for system `#include` directories is the system include directory and any directories specified with the `-I` option. The path for user `#include` directives is the location of the file containing the `#include` directive, followed by the path specified for the system `#include` directive.

By default, the text in files included with an `#include` directive is not included in the client and server code that is generated.

`-D`*`symbol`*

Specifies a symbol to be defined during OMG IDL file preprocessing. The `m3idltojava` command passes this symbol to the preprocessor.

`-U`*`symbol`*

Specifies a symbol to be undefined during OMG IDL file preprocessing. The `m3idltojava` command passes this symbol to the preprocessor.

`-f`*`options`*

You can enable the following options by specifying them as shown, and disable them by appending the string `no-`. For example, to prevent the C preprocessor from being run on the input OMG IDL files, specify `-fno-cpp`.

`-flist-flags`

Displays the state of all -f flags. By default, this option is disabled.

`-fclient`

Generates the client application files. By default, this option is enabled.

`-fserver`
: Generates the server application files. By default, this option is enabled.

`-fverbose`
: Specifies that the `m3idltojava` command should work in verbose mode. In particular, it writes command output to its standard output. By default, this option is disabled.

`-fversion`
: Specifies that the compiler prints its version and timestamp. By default, this option is disabled.

Examples The following command generates only the server application files for `Simple.idl`:

```
m3idltojava -fno-client Simple.idl
```

The following command generates only the client application files for `Simple.idl`:

```
m3idltojava -fno-server Simple.idl
```

CHAPTER

11 CORBA ORB

This chapter supplements the information in package `org.omg.CORBA` by providing the following topics:

- Initializing the ORB, which includes the section “Passing the Address of the IIOP Listener”
- Initializing the ORB for Native and Remote Clients

Note: For details about the API for package `org.omg.CORBA`, see the Java IDL document published by the Sun Microsystems, Inc. and distributed with the JDK 1.2.

Initializing the ORB

[This section is reprinted from the package information for `org.omg.CORBA`, as published by Sun Microsystems, Inc. for the JDK 1.2.]

An application or applet gains access to the CORBA environment by initializing itself into an ORB using one of three `init` methods. Two of the three methods use the properties (associations of a name with a value) shown in the following table:

Property Name	Property Value
`org.omg.CORBA.ORBClass`	Class name of an ORB implementation
`org.omg.CORBA.ORBSingletonClass`	Class name of the ORB returned by `init()`

These properties allow a different vendor's ORB implementation to be "plugged in."

When an ORB instance is being created, the class name of the ORB implementation is located using the following standard search order:

1. Check in Applet parameter or application string array, if any.
2. Check in properties parameter, if any.
3. Check in the System properties (currently applications only).
4. Fall back on a hardcoded default behavior (use the Java IDL implementation).

Note that the WebLogic Enterprise ORB provides a default implementation for the fully functional ORB and for the Singleton ORB. When the `init` method is given no parameters, the default Singleton ORB is returned. When the `init` method is given parameters but no ORB class is specified, the Java IDL ORB implementation is returned.

The following code fragment creates an ORB object initialized with the default ORB Singleton. This ORB has a restricted implementation to prevent malicious applets from doing anything beyond creating typecodes. It is called a Singleton because there is only one instance for an entire virtual machine.

```
ORB orb = ORB.init();
```

The following code fragment creates an ORB object and a Singleton ORB object for an application.

```
Properties p = new Properties();
p.put("org.omg.CORBA.ORBClass", "com.sun.CORBA.iiop.ORB");
p.put("org.omg.CORBA.ORBSingletonClass","com.sun.CORBA.idl.ORBSingleton");
System.setProperties(p);
ORB orb = ORB.init(args, p);
```

In the preceding code fragment, note the following:

- ♦ The ORB class is to be initialized as `com.sun.CORBA.iiop.ORB`.
- ♦ The SingletonORB class is to be initialized as `com.sun.CORBA.idl.ORBSingleton`.
- ♦ The statement `System.setProperties(p)` sets the system properties based on the value of `p`.
- ♦ The parameter `args` represents the arguments supplied to the application's main method. If `p` is `null`, and the arguments do not specify an ORB class, the new ORB is initialized with the default Java IDL implementation.

Note: Due to the security restrictions on applets, you will probably not be able to invoke the `System.setProperties` method from within an applet. Instead, you can set the `org.omg.CORBA.ORBClass` and `org.omg.CORBA.ORBSingletonClass` parameters via HTML before starting the applet.

The following code fragment creates an ORB object for the applet supplied as the first parameter. If the given applet does not specify an ORB class, the new ORB will be initialized with the default WebLogic Enterprise ORB implementation.

```
ORB orb = ORB.init(myApplet, null);
```

An application or applet can be initialized in one or more ORBs. ORB initialization is a bootstrap call into the CORBA world.

Passing the Address of the IIOP Listener

When you compile WebLogic Enterprise client and server applications, use the `-DTOBJADDR` option to specify the host and port of the IIOP Listener. This allows you, in the application code, to specify `null` as a host and port string in invocations to:

- The `ORB.init` method
- The local Bootstrap object

By keeping host and port specifications out of your client and server application code, you maximize the portability and reusability of your application code.

Initializing the ORB for Native and Remote Clients

WebLogic Enterprise provides two methods on the `com.beasys.Tobj_Bootstrap` object that client applications use to initialize the ORB, depending on whether the client is *native* (that is, on a process that is inside the WebLogic Enterprise domain) or *remote* (that is, on a machine that needs to communicate to the server application via the IIOP Listener/Handler).

These two methods are:

- The `getNativeProperties` method

  This method returns a set of properties that need to be passed in a subsequent invocation of the `org.omg.CORBA.ORB.init` method. This subsequent invocation causes BEA's Java ORB to be initialized. The `getNativeProperties` method also initializes the WebLogic Enterprise infrastructure.

  The `getNativeProperties` method must be invoked before any attempt is made to access any class in the `org.omg.CORBA` package; otherwise, errors will occur when receiving CORBA exceptions from the server.

- The `getRemoteProperties` method

  This method returns the properties needed to initialize the ORB for remote clients. The `getRemoteProperties` method is specified for symmetry and always returns `null`.

Note: In WLE 4.2, Java native clients are not supported.

CHAPTER

12 Mapping IDL to Java

This chapter contains the following topics:

- IDL to Java Overview
- Package Comments on Holder Classes
- Exceptions. This section includes the following topics:
  - Differences Between CORBA and Java Exceptions
  - System Exceptions
  - User Exceptions
  - Minor Code Meanings

Note: This chapter contains excerpts from the Java IDL document published by Sun Microsystems, Inc. and distributed with the JDK 1.2.

IDL to Java Overview

The `idltojava` and `m3idltojava` tools read an OMG IDL interface and translate it, or map it, to a Java interface. The `m3idltojava` tool also creates stub, skeleton, helper, holder, and other files as necessary. While the `idltojava` tool creates stub, skeleton, helper, holder, and other files, the skeleton files it produces cannot be used with the WebLogic Enterprise system. When compiling the OMG IDL files to build server skeletons to be used with the WebLogic Enterprise system, be sure to use the `m3idltojava` tool.

These `.java` files are generated from the OMG IDL file according to the mapping specified in the OMG document *IDL/Java Language Mapping* (available from the OMG Web site at `http://www.omg.org`). We cross-reference the following four chapters of that document here for your convenience:

- Chapter 5, “Mapping IDL to Java”
- Chapter 6, “Mapping Pseudo-Objects to Java”
- Chapter 7, “Server-Side Mapping”
- Chapter 8, “Java ORB Portability Interfaces”

A summary of the IDL to Java language mapping follows.

CORBA objects are defined in OMG IDL. Before they can be used by a Java programmer, their interfaces must be mapped to Java classes and interfaces. Sun Microsystems, Inc. provides the `idltojava` tool, and the WebLogic Enterprise system includes the `m3idltojava` tool, which performs this mapping automatically.

This overview shows the correspondence between OMG IDL constructs and Java constructs. Note that OMG IDL, as its name implies, defines interfaces. Like Java interfaces, IDL interfaces contain no implementations for their operations (methods in Java). In other words, IDL interfaces define only the signature for an operation (the name of the operation, the datatype of its return value, the datatypes of the parameters that it takes, and any exceptions that it raises). The implementations for these operations need to be supplied in Java classes written by a Java programmer.

The following table lists the main constructs of IDL and the corresponding constructs in Java.

IDL Construct	**Java Construct**
`module`	`package`
`interface`	`interface, helper class, holder class`
`constant`	`public static final`
`boolean`	`boolean`
`char, wchar`	`char`
`octet`	`byte`

IDL Construct	**Java Construct**
`string, wstring`	`java.lang.String`
`short, unsigned short`	`short`
`long, unsigned long`	`int`
`long long, unsigned long long`	`long`
`float`	`float`
`double`	`double`
`enum, struct, union`	`class`
`sequence, array`	`array`
`exception`	`class`
`readonly attribute`	method for accessing value of attribute
`readwrite attribute`	methods for accessing and setting value of attribute
`operation`	method

Note: When a CORBA operation takes a type that corresponds to a Java object type (a `String`, for example), it is illegal to pass a Java `null` as the parameter value. Instead, pass an empty version of the designated object type (for example, an empty `String` or an empty array). A Java `null` can be passed as a parameter only when the type of the parameter is a CORBA object reference, in which case the `null` is interpreted as a `nil` CORBA object reference.

Package Comments on Holder Classes

Operations in an IDL interface may take `out` or `inout` parameters, as well as `in` parameters. The Java programming language only passes parameters by value and thus does not have `out` or `inout` parameters; therefore, these are mapped to what are called Holder classes. In place of the IDL `out` parameter, the Java programming language method will take an instance of the Holder class of the appropriate type. The result that was assigned to the `out` or `inout` parameter in the IDL interface is assigned to the value field of the Holder class.

The package `org.omg.CORBA` contains a Holder class for each of the basic types (`BooleanHolder`, `LongHolder`, `StringHolder`, and so on). It also has Holder classes for each generated class (such as `TypeCodeHolder`), but these are used transparently by the ORB, and the programmer usually does not see them.

The Holder classes defined in the package `org.omg.CORBA` are:

```
AnyHolder
BooleanHolder
ByteHolder
CharHolder
DoubleHolder
FloatHolder
IntHolder
LongHolder
ObjectHolder
PrincipalHolder
ShortHolder
StringHolder
TypeCodeHolder
```

Exceptions

CORBA has two types of exceptions: standard system exceptions, which are fully specified by OMG, and user exceptions, which are defined by the individual application programmer. CORBA exceptions are a little different from Java exception objects, but those differences are largely handled in the mapping from IDL to Java.

Topics in this section include:

- Differences Between CORBA and Java Exceptions
- System Exceptions, which includes the following subtopics:
  - System Exception Structure
  - Minor Codes
  - Completion Status

- User Exceptions
- Minor Code Meanings

Differences Between CORBA and Java Exceptions

To specify an exception in IDL, the interface designer uses the `raises` keyword. This is similar to the `throws` specification in Java. When you use the exception keyword in IDL, you create a user-defined exception. The standard system exceptions cannot be specified this way.

System Exceptions

CORBA defines a set of standard system exceptions, which are generally raised by the ORB libraries to signal systemic error conditions like:

- Server-side system exceptions, such as resource exhaustion or activation failure
- Communication system exceptions, such as losing contact with the object, host down, or cannot talk to ORB daemon (`orbd`)
- Client-side system exceptions, such as invalid operand type or anything that occurs before a request is sent or after the result comes back

All IDL operations can throw system exceptions when invoked. The interface designer need not specify anything to enable operations in the interface to throw system exceptions -- the capability is automatic.

This makes sense because no matter how trivial an operation’s implementation is, the potential of an operation invocation coming from a client that is in another process, and perhaps (likely) on another machine, means that a whole range of errors is possible.

Therefore, a CORBA client should always catch CORBA system exceptions. Moreover, developers cannot rely on the Java compiler to notify them of a system exception they should catch, because CORBA system exceptions are descendants of `java.lang.RuntimeException`.

System Exception Structure

All CORBA system exceptions have the same structure:

```
exception <SystemExceptionName> { // descriptive of error
    unsigned long minor;          // more detail about error
    CompletionStatus completed;   // yes, no, maybe
}
```

System exceptions are subtypes of `java.lang.RuntimeException` through `org.omg.CORBA.SystemException`:

```
java.lang.Exception
 |
 +--java.lang.RuntimeException
     |
     +--org.omg.CORBA.SystemException
         |
         +--BAD_PARAM
         |
         +--//etc.
```

Minor Codes

All CORBA system exceptions have a minor code field, which contains a number that provides additional information about the nature of the failure that caused the exception. Minor code meanings are not specified by the OMG; each ORB vendor specifies appropriate minor codes for that implementation. For the meaning of minor codes thrown by the Java ORB, see the section “Minor Code Meanings.”

Completion Status

All CORBA system exceptions have a completion status field, which indicates the status of the operation that threw the exception. The completion codes are:

`COMPLETED_YES`	The object implementation has completed processing prior to the exception being raised.
`COMPLETED_NO`	The object implementation was not invoked prior to the exception being raised.
`COMPLETED_MAYBE`	The status of the invocation is unknown.

User Exceptions

CORBA user exceptions are subtypes of `java.lang.Exception` through `org.omg.CORBA.UserException`:

```
java.lang.Exception
 |
 +--org.omg.CORBA.UserException
      |
      +-- Stocks.BadSymbol
      |
      +--//etc.
```

Each user-defined exception specified in IDL results in a generated Java exception class. These exceptions are entirely defined and implemented by the programmer.

Minor Code Meanings

System exceptions all have a field minor that allows CORBA vendors to provide additional information about the cause of the exception. As stated in the CORBA 2.2 specification (13.4.2 Reply Message), the high order 20 bits of minor code value contain a 20-bit "vendor minor codeset ID" (VMCID); the low order 12 bits contain a minor code. BEA's VMCID is 0x54555000. Further, Sun defines single or double digit minor codes for its Java IDL ORB and BEA defines its minor code starting from 1,000. Thus, a condition common to either ORB uses the Java IDL minor code (and VMCID 0), and the BEA ORB unique minor code is 1,000 or greater.

For Sun Microsystems, Inc. minor codes, see the Java IDL documentation. For BEA's minor codes, see the *Release Notes*.

Table 12-1 ORB Minor Codes and Their Meanings

Code	Meaning
	BAD_PARAM Exception Minor Codes
1	A null parameter was passed to a Java IDL method.
	COMM_FAILURE Exception Minor Codes

Code	Meaning
1	Unable to connect to the host and port specified in the object reference, or in the object reference obtained after location/object forward.
2	Error occurred while trying to write to the socket. The socket has been closed by the other side, or is aborted.
3	Error occurred while trying to write to the socket. The connection is no longer alive.
6	Unable to successfully connect to the server after several attempts.
	DATA_CONVERSION Exception Minor Codes
1	Encountered a bad hexadecimal character while doing ORB `string_to_object` operation.
2	The length of the given IOR for `string_to_object()` is odd. It must be even.
3	The string given to `string_to_object()` does not start with `IOR:` and hence is a bad stringified IOR.
4	Unable to perform ORB `resolve_initial_references` operation due to the host or the port being incorrect or unspecified, or the remote host does not support the Java IDL bootstrap protocol.
	INTERNAL Exception Minor Codes
3	Bad status returned in the IIOP Reply message by the server.
6	When unmarshaling, the repository id of the user exception was found to be of incorrect length.
7	Unable to determine local hostname using the Java API's `InetAddress.getLocalHost().getHostName().`
8	Unable to create the listener thread on the specific port. Either the port is already in use, there was an error creating the daemon thread, or security restrictions prevent listening.
9	Bad locate reply status found in the IIOP locate reply.
10	Error encountered while stringifying an object reference.
11	IIOP message with bad GIOP v1.0 message type found.
14	Error encountered while unmarshaling the user exception.

Code	Meaning
18	Internal initialization error.
	INV_OBJREF Exception Minor Codes
1	An IOR with no profile was encountered.
	MARSHAL Exception Minor Codes
4	Error occured while unmarshaling an object reference.
5	Marshaling/unmarshaling unsupported IDL types like wide characters and wide strings.
6	Character encountered while marshaling or unmarshaling a character or string that is not ISO Latin-1 (8859.1) compliant. It is not in the range of 0 to 255.
	NO_IMPLEMENT Exception Minor Codes
1	Dynamic Skeleton Interface is not implemented.
	OBJ_ADAPTER Exception Minor Codes
1	No object adapter was found matching the one in the object key when dispatching the request on the server side to the object adapter layer.
2	No object adapter was found matching the one in the object key when dispatching the locate request on the server side to the object adapter layer.
4	Error occured when trying to connect a servant to the ORB.
	OBJ_NOT_EXIST Exception Minor Codes
1	Locate request got a response indicating that the object is not known to the locator.
2	Server id of the server that received the request does not match the server id baked into the object key of the object reference that was invoked upon.
4	No skeleton was found on the server side that matches the content of the object key inside the object reference.
	UNKNOWN Exception Minor Codes
1	Unknown user exception encountered while unmarshaling: the server returned a user exception that does not match any expected by the client.

Code	Meaning
3	Unknown run-time exception thrown by the server implementation.

Table 12-2 Name Server Minor Codes and Their Meanings

Code	Meaning
	INITIALIZE Exception Minor Codes
150	Transient name service caught a `SystemException` while initializing.
151	Transient name service caught a Java exception while initializing.
	INTERNAL Exception Minor Codes
100	An `AlreadyBound` exception was thrown in a `rebind` operation.
101	An `AlreadyBound` exception was thrown in a `rebind_context` operation.
102	Binding type passed to the internal binding implementation was not `BindingType.nobject` or `BindingType.ncontext`.
103	Object reference was bound as a context, but it could not be narrowed to `CosNaming.NamingContext`.
200	Implementation of the `bind` operation encountered a previous binding.
201	Implementation of the `list` operation caught a Java exception while creating the list iterator.
202	Implementation of the `new_context` operation caught a Java exception while creating the new `NamingContext` servant.
203	Implementaton of the `destroy` operation caught a Java exception while disconnecting from the ORB.