BEA WebLogic Enterprise

CORBA Java Programming Reference

WebLogic Enterprise 5.1
Document Edition 5.1
May 2000

Copyright

Restricted Rights Legend

Trademarks or Service Marks

CORBA Java Programming Reference

Document Edition	Date	Software Version
5.1	May 2000	BEA WebLogic Enterprise 5.1

Contents

4. Java Bootstrap Object Programming Reference

11. Joint Client/Server Applications

12. Java Development and Administration Commands

13. CORBA ORB

14. Mapping IDL-to-Java

About This Document

This document describes the BEA WebLogic Enterprise™ CORBA Java application programming interface (API).

This document covers the following topics:

- Chapter 1, “OMG IDL Syntax,” describes the Object Management Group (OMG) Interface Definition Language (IDL) and OMG IDL extensions.
- Chapter 2, “Server Description File,” describes the Server Description File.
- Chapter 3, “Java TP Framework,” describes the WebLogic Enterprise TP Framework application programming interface (API).
- Chapter 4, “Java Bootstrap Object Programming Reference,” describes the Bootstrap object.
- Chapter 5, “FactoryFinder Interface,” describes the FactoryFinder interface.
- Chapter 6, “Security Service,” directs you to information about the Security Service.
- Chapter 7, “Transactions Service,” directs you to information about the Transactions Service.
- Chapter 8, “Notification Service,” directs you to information about the Notification Service.
- Chapter 9, “Request-Level Interceptors,” directs you to information about Request-Level Interceptors.
- Chapter 10, “Interface Repository Interfaces,” describes the Interface Repository interfaces.

- Chapter 11, “Joint Client/Server Applications,” describes how to program joint client/server applications and the BEAWrapper Callbacks API.
- Chapter 12, “Java Development and Administration Commands,” directs you to information about the build and administration commands for UNIX and Windows NT platforms.
- Chapter 13, “CORBA ORB,” provides supplemental information about the CORBA ORB.
- Chapter 14, “Mapping IDL-to-Java,” describes the 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)

What You Need to Know

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 CORBA *C++ Programming Reference* in the WebLogic Enterprise online documentation.

e-docs Web Site

The BEA WebLogic Enterprise product documentation is available on the BEA System, Inc. corporate Web site. From the BEA Home page, click the Product Documentation button or go directly to the "e-docs" Product Documentation page at http://e-docs.bea.com.

How to Print the Document

You can print a copy of this document from a Web browser, one file at a time, by using the File—>Print option on your Web browser.

A PDF version of this document is available on the WebLogic Enterprise documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Enterprise documentation Home page, click the PDF Files button, and select the document you want to print.

If you do not have Adobe Acrobat Reader installed, you can download it for free from the Adobe Web site at http://www.adobe.com/.

Related Information

For more information about CORBA, Java 2 Enterprise Edition (J2EE), BEA Tuxedo®, distributed object computing, transaction processing, C++ programming, and Java programming, see the *BEA WebLogic Enterprise Bibliography* in the WebLogic Enterprise online documentation.

Contact Us!

Your feedback on the BEA WebLogic Enterprise documentation is important to us. Send us e-mail at **docsupport@bea.com** if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the WebLogic Enterprise documentation.

In your e-mail message, please indicate that you are using the documentation for the BEA WebLogic Enterprise 5.1 release.

If you have any questions about this version of BEA WebLogic Enterprise, or if you have problems installing and running BEA WebLogic Enterprise, contact BEA Customer Support through BEA WebSUPPORT at www.bea.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

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.

Convention	Item
italics	Indicates emphasis or book titles.

`monospace text`	Indicates code samples, commands and their options, data structures and their members, data types, directories, and filenames 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 *`
	`\tux\data\ap`
	`.doc`
	`tux.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. *Example*s: 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`*`]...`

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

. . .	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.

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.

Note: When using a pragma version statement, be sure to locate it after the corresponding interface definition. Here is an example of proper usage:

```
module A
{
    interface B
    {
#pragma version B "3.5"
         void op1();
    };
};
```

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."

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.

2 Server Description File

This topic includes the following sections:

- Creating the Server Description File. This section includes:
  - 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 BEA 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 BEA WebLogic Enterprise system is packaged with the BEA 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 BEA 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 BEA 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 BEA 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 BEA 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 Reference Manual*) 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 BEA 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 API Javadoc.

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 pathname. 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 CORBA Java Server Applications*.

CHAPTER

3 Java TP Framework

This topic includes the following sections:

- A Simple Programming Model. This section describes:
  - Control Flow
  - Object State Management
  - Transaction Integration
  - Object Housekeeping
  - High-level Services
- Object State Management. This section describes:
  - Activation Policy
  - Application-controlled Activation and Deactivation
  - Saving and Restoring Object State
- Transactions. This section describes:
  - Transaction Policies
  - Transaction Initiation
  - Transaction Termination
  - Transaction Suspend and Resume
  - Restrictions on Transactions
  - Voting on Transaction Outcome
  - Transaction Time-outs
- Java TP Framework Interfaces. This section describes:

- Tobj_Servant Interface
- Server Object
- TP Interface

- Error Conditions and Exceptions. This section describes:
  - Exceptions Raised by the Java TP Framework
  - Exceptions in the Server Application Code
  - Exceptions and Transactions

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

BEA WebLogic Enterprise uses BEA Tuxedo as the underlying infrastructure for providing load balancing, transactional capabilities, and administrative infrastructure. The base API used by the TP Framework is the CORBA API with BEA extensions.

Before BEA WebLogic Enterprise, ORB products did not approach BEA Tuxedo's performance in large-scale environments. BEA Tuxedo systems support applications that can process hundreds of transactions per second. These applications are built using the BEA Tuxedo stateless-service programming model that minimizes the amount of system resources used for each request, and thus maximizes throughput and price performance.

Now, BEA WebLogic Enterprise and its Java TP Framework let you develop CORBA applications with performance similar to BEA Tuxedo applications. BEA WebLogic Enterprise servers that use the Java TP Framework provide throughput, response time, and price performance approaching the BEA Tuxedo stateless-service programming model, while using the CORBA programming model.

The Java 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
  - Create object references and preactivate 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
  - Obtain object IDs in object references that were created in the Java TP Framework
  - Open and close XA resource managers
  - Log messages to a user log (`ULOG`) file
  - Obtain object references to the ORB and to Bootstrap objects

A Simple Programming Model

The Java TP Framework provides a simple, useful subset of the wide range of possible CORBA object implementation choices. You use it for the development of server-side object implementations only.

When using any client-side CORBA ORB, clients interact with CORBA objects whose server-side implementations are managed by the Java TP Framework. Clients are unaware of the existence of the TP Framework—a client written to access a CORBA object executing in a non-BEA BEA WebLogic Enterprise server environment will be able to access that same CORBA object executing in a BEA WebLogic Enterprise server environment without any changes or restrictions to the client interface.

The Java TP Framework provides a server environment and an API that is easier to use and understand than the CORBA Portable Object Adapter (POA) API, and is specifically geared towards enterprise applications. It is a simple server programming model and an orthodox implementation of the CORBA model, which will be familiar to programmers using ORBs such as ORBIX or VisiBroker.

The Java TP Framework simplifies the programming of BEA WebLogic Enterprise servers by reducing the complexity of the server environment in the following ways:

- The Java TP Framework does all interactions with the POA and the naming service. The application programmer requires no knowledge of the POA or naming service interfaces.
- A CORBA object may be involved in only one transaction at a time (consistent with the association of one object ID to one servant).

The Java TP Framework provides the following functionality:

- Control Flow
- Object State Management
- Transaction Integration
- Object Housekeeping
- High-level Services

The TP Framework API is available for use in either a single threaded or multi-threaded Java server.

Control Flow

The Java TP Framework, in conjunction with the ORB and the POA, controls the flow of the application program by doing the following:

- Controlling the server mainline and invoking callback methods on classes defined by the TP Framework at appropriate times for server startup and shutdown. This relieves the application programmer from complex interactions related to ORB and POA initialization and coordination of transactions, resource managers, and object state on shutdown.
- Scheduling objects for activation and deactivation when client requests arrive and are completed. This removes the complexity of management of object activation and deactivation from the realm of the application programmer and enables the use of the TP monitor infrastructure's powerful load-balancing capabilities, crucial to performance of mission-critical tasks.

Object State Management

The Java TP Framework API provides callback methods for application code to implement flexible state management schemes for CORBA objects. State management involves the saving and restoring of object state on object deactivation and activation. State management also concerns the duration of activation of objects, which influences the performance of servers and their resource usage. The default duration of object activation is controlled by policies assigned to implementations at IDL compile time. For more information about object state management, see the section "Object State Management" on page 3-6.

Transaction Integration

Java TP Framework transaction integration provides the following features:

- CORBA objects can participate in global transactions.
- Objects participating in transactions can be implemented as stateful objects that remain in memory for the duration of a transaction (by using the transaction activation policy) to decrease client response time.
- CORBA objects that participate in transactions can affect transaction outcome either during their transactional work or just prior to the system's execution of the two-phase commit algorithm after all transactional work has been completed.
- Transactions can be automatically initiated on the server, which is transparent to the client.

Object Housekeeping

When a server is shut down, the Java TP Framework rolls back any transactions that the server is involved in and deactivates any CORBA objects that are currently active.

High-level Services

The TP interface in the Java TP Framework API provides methods for performing object registrations and utility functions. The following services are provided:

- Object reference creation
- Factory-based routing support
- Accessors for system objects, such as the ORB
- Registration and unregistration of factories with the Factory Finder
- Application-controlled activation and deactivation
- User logging

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) and the BEA Tuxedo system. By encapsulating the underlying API calls with a high-level set of methods, programmers can focus their efforts on providing business logic, rather than on understanding and using the more complex underlying facilities.

Object State Management

Object state management involves the saving and restoring of object state on object deactivation and activation. It also concerns the duration of activation of objects, which influences the performance of servers and their resource usage. The external API of the Java TP Framework provides `activate_object` and `deactivate_object` methods, which are a possible location for state management code.

Activation Policy

State management is provided in the TP Framework by the activation policy. This policy controls the activation and deactivation of servants for a particular IDL interface (as opposed to the creation and destruction of the servants). This policy is applicable only to CORBA objects using the Java TP Framework.

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. You can choose from one of three activation policies: `method` (the default), `transaction`, or `process`.

Note: The activation policies are set in an Server Description file that is configured at OMG IDL compile time. For a description of the Server Description file, refer to Chapter 2, "Server Description File."

The activation policies are described below:

- `method` (This is the default activation policy.)

  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. The object is activated before the first method invocation on the object and is deactivated in one of the following ways:

  - If a user-initiated transaction is in effect when the object is activated, the object is deactivated when the first of the following occurs: the transaction is committed or rolled back, or the server is shut down in an orderly fashion. The latter is done using either the `tmshutdown`(1) or `tmadmin`(1) command. These commands are described in the *BEA Tuxedo Reference Manual* online document.

- If a user-initiated transaction is not in effect when the TP object is activated, the TP object is deactivated when the method completes.

The `transaction` activation policy provides a means for an object to vote on the outcome of the transaction prior to the execution of the two-phase commit algorithm. An object votes to roll back the transaction by calling `Current.rollback_only()` in the com.beasys.`Tobj_ServantBase.deactivate_object` method. It votes to commit the transaction by not calling `Current.rollback_only()` in the method.

Note: This is a model of resource allocation that is similar to that of a BEA Tuxedo conversational service. However, this model is less expensive than the BEA Tuxedo conversational service in that it uses fewer system resources. This is because of the BEA 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 that 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 begins when it is invoked while in an inactive state and, by default, lasts until the end of the process.

Note: The Java TP Framework API provides an interface method (com.beasys.`TP.deactivateEnable`) that allows the application to control the timing of object deactivation for objects that have the `activation policy` set to `process`.

The TP Framework API also provides an interface method (`com.beasys.TP.create_active_object_reference`) that allows the application to pre-activate the CORBA object at the time that its object reference is created.

Application-controlled Activation and Deactivation

Ordinarily, activation and deactivation decisions are made by the Java TP Framework, as discussed earlier in this chapter. The techniques in this section show how to use alternate mechanisms. The application can control the timing of activation and deactivation explicitly for objects with particular policies.

Explicit Activation

Application code can bypass the on-demand activation feature of the Java TP Framework for objects that use the `process` activation policy. The application can "preactivate" an object (that is, activate it before any invocation) using the `com.beasys.TP.create_active_object_reference` call.

Preactivation works as follows. Before the application creates an object reference, the application instantiates a servant and initializes that servant's state. The application uses `com.beasys.TP.create_active_object_reference` to put the object into the Active Object Map (that is, associate the servant with an `ObjectId`). Then, when the first invocation is made, the Java TP Framework immediately directs the request to the process that created the object reference and then to the existing servant, bypassing the call to the servant's `activate_object` method (just as if this were the second or later invocation on the object). Note that the object reference for such an object will not be directed to another server and the object will never go through on-demand activation as long as the object remains activated.

Since the preactivated object has the `process` activation policy, it will remain active until one of two events occurs: (1) the ending of the process or (2) a `com.beasys.TP.deactivateEnable` call.

Usage Notes

Preactivation is especially useful if the application needs to establish the servant with an initial state in the same process, perhaps using shared memory to initialize state. Waiting to initialize state until a later time and in a potentially different process may be very difficult if that state includes pointers, object references, or complex data structures. `com.beasys.TP.create_active_object_reference` guarantees that the preactivated object is in the same process as the code that is doing the preactivation. While this is convenient, preactivation should be used sparingly, as should all process objects, because it preallocates precious resources. However, when needed and used properly, preallocation is more efficient than alternatives.

Examples of such usage might be an object using the "iterator" pattern. For example, there might a potentially long list of items that could be returned (in an unbound IDL sequence) from a "database_query" method (for example, the contents of the telephone book). Returning all such items in the sequence is impractical because the message size and the memory requirements would be too large.

On an initial call to get the list, an object using the iterator pattern returns only a limited number of items in the sequence and also returns a reference to an "iterator" object that can be invoked to receive further elements. This iterator object is initialized by the initial object; that is, the initial object creates a servant and sets its state to keep track of where in the long list of items the iteration currently stands (the pointer to the database, the query parameters, the cursor, and so forth).

The initial object uses `com.beasys.TP.create_active_object_reference` to preactivate this iterator object and to create its reference which will be returned to the client. It also creates an object reference to that object to return to the client. The client then invokes repeatedly on the iterator object to receive, say, the next 100 items in the list each time. The advantage of preactivation in this situation is that the state might be complex. It is often easiest to set such state initially, from a method that has all the information in its context (call frame), when the initial object still has control.

When the client is finished with the iterator object, it invokes a final method on the initial object, which deacativates the iterator object. The initial object deactivates the iterator object by invoking a method on the iterator object that calls the `com.beasys.TP.deactivateEnable` method; that is, the iterator object calls `com.beasys.TP.deactivateEnable` on itself.

Caution

For objects to be preactivated in this fashion, the state usually cannot be recovered if a crash occurs. (This is because the state was considered too complex or inconvenient to set upon initial, delayed activation.) This is a valid object technique, essentially stating that the object is valid only for a single activation period.

However, a problem may arise because of the "one-time" usage. Since a client still holds an object reference that leads to the process containing that state, and since the state cannot be recreated after the crash, care must be taken that the client's next invocation does not automatically provoke a new activation of the object, because that object would have inapplicable state.

The solution is to refuse to allow the object to be activated automatically by the TP Framework. If the activate_object method throws a `com.beasys.TobjS.ActivateObjectFailed` exception, the TP Framework will not complete the activation and will return the org.omg.`CORBA.OBJECT_NOT_EXIST` exception to the client. The client has presumably been warned about this possibility, since it knows about the iterator (or similar) pattern. The client must be prepared to restart the iteration.

Self-deactivation

Just as it is possible to preactivate an object with the `process` activation policy, it is possible to request the deactivation of an object with the `process` activation policy. The ability to preactivate and the ability to request deactivation are independent; regardless of how an object was activated, it can be deactivated explicitly.

A method in the application can request (via `com.beasys.TP.deactivateEnable`) that the object be deactivated. When `com.beasys.TP.deactivateEnable` is called and the object is subsequently deactivated, no guarantee is made that subsequent invocations on the CORBA object will result in reactivation in the same process as a previous activation. The association between the `ObjectId` and the servant exists from the activation of the CORBA object until one of the following events occurs: (1) the shutdown of the server process or (2) the application calls `com.beasys.TP.deactivateEnable`. After the association is broken, when the object is invoked again, it can be re-activated anywhere that is allowed by the BEA WebLogic Enterprise configuration parameters.

When a `com.beasys.TP.deactivateEnable` call is invoked, the object currently executing is deactivated after completion of the method in which the call is made. The object itself makes the decision that it should be deactivated. This is often done during a method call that acts as a "signoff" signal.

Note: The `TP::deactivateEnable(interface, object id, servant)` `method` can be used to deactivate an object. However, if that object is currently in a transaction, the object will be deactivated when the transaction commits or rolls back. If an invoke occurs on the object before the transaction is committed or rolled back, the object will not be deactivated.

To ensure the desired behavior, make sure that the object is not in a transaction or ensure that no invokes occur on the object after the `TP::deactivateEnable()` call until the transaction is complete.

Saving and Restoring Object State

While CORBA objects are active, their state is contained in a servant. Unless an application uses `com.beasys.TP.create_active_object_reference`, state must be initialized when the object is 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 they have been deactivated. While a CORBA object is deactivated, its state must be saved outside the process in which the servant was active. The object's state can be saved in shared memory, in a file, or in a database. Before a CORBA object is deactivated, its state must be saved; when it is activated, its state must be restored.

The programmer determines what constitutes an object's state and what must be saved before an object is deactivated, and restored when an object is activated.

Use of Constructors for Java Corba Objects

The state of Java CORBA objects must not be initialized in the constructors for the servant classes. This is because the Java TP Framework may reuse an instance of a servant. No guarantee is made as to the timing of the creation of servant instances.

Transactions

The following sections provide information about transaction policies and how to use transactions.

Transaction Policies

Eligibility of CORBA objects to participate in global transactions is controlled by the transaction policies assigned to implementations at compile time. The following policies can be assigned.

Note: The activation policies are set in an Server Description file that is configured at OMG IDL compile time. For a description of the Server Description file, refer to Chapter 2, "Server Description File."

- `never`

  The implementation is not transactional. Objects created for this interface can never be involved in 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. This policy instructs the system to allow requests within a transaction to be made of this implementation. An `AUTOTRAN` policy specified in the `UBBCONFIG` file for the interface is ignored.

- `optional` (This is the default `transaction_policy`.)

  The implementation may be transactional. Objects can be involved in a transaction if the request is transactional. Servers containing transactional objects must be configured within a group associated with an XA-compliant resource manager. If the `AUTOTRAN` parameter is specified in the `UBBCONFIG` file for the interface, `AUTOTRAN` is on.

- `always`

  The implementation is transactional. Objects are required to always be involved in a transaction. If a request is made outside a transaction, the system automatically starts a transaction before invoking the method. The transaction is committed when the method ends. (This is the same behavior that results from specifying `AUTOTRAN` for an object with the option transaction policy, except that no administrative configuration is necessary to achieve this behavior, and it cannot be overridden by administrative configuration.) Servers containing transactional objects must be configured within a group that is associated with an XA-compliant resource manager.

Note: The `optional` policy is the only transaction policy that can be influenced by administrative configuration. If the system administrator sets the `AUTOTRAN` attribute for the interface by means of the `UBBCONFIG` file or by using administrative tools, the system automatically starts a transaction upon invocation of the object, if it is not already infected with a transaction (that is, the behavior is as if the `always` policy were specified).

Transaction Initiation

Transactions are initiated in one of two ways:

- By the application code via use of the `org.omg.CosTransactions.Current.begin` method. This can be done in either the client or the server. For a description of this operation, see *Using Transactions*.
- By the system when an invocation is done on an object that has either:
  - Transaction policy `always`
  - Transaction policy `optional` and a setting of `AUTOTRAN` for the interface

  For more information, refer to the *Administration Guide*.

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 Java TP Framework never commits a transaction. The BEA 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 BEA WebLogic Enterprise system.

The following behavior is enforced by the BEA 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 Java 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, the object 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 BEA 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 Java 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 Java TP Framework raises the `org.omg.CORBA.OBJ_ADAPTER` exception to the client application and the transaction is rolled back by the BEA 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 on Transactions

The following restrictions apply to BEA WebLogic Enterprise transactions:

- A CORBA object in the BEA 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 Java 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 Java 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: Users of SQL cursors must be careful when using an object with the `method` or `process` activation policy. A typical operation would be for a process to open an SQL cursor within a client-initiated transaction. For typical SQL database products, once the client commits the transaction, all cursors that were opened within that transaction are automatically closed; however, the object will not receive any notification that its cursor has been closed.

Transaction Time-outs

When a transaction time-out occurs, the transaction is marked so that the only possible outcome is to roll back the transaction, and the `org.omg.CORBA.TRANSACTION_ROLLEDBACK` standard exception is returned to the client. Any attempts to send new requests will also fail with the `org.omg.CORBA.TRANSACTION_ROLLEDBACK` exception until the transaction has been aborted.

Java TP Framework Interfaces

The Java TP Framework supports the following interfaces:

- `com.beasys.Tobj_Servant`
- `com.beasys.Tobj.Server`
- `com.beasys.Tobj.TP`

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 Java 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 Java TP Framework guarantees will be invoked for CORBA object activation and deactivation. The servant class constructor may or may not be invoked at activation time. Therefore, the server-application code must not do any state handling for CORBA objects in the constructor of the servant class.

Server Object

The `com.beasys.Tobj.Server` object provides default callback methods to initialize and release the server application. A new class that derives from the `com.beasys.Tobj.Server` class can be implemented that overrides the `initialize` and `release` methods with application-specific server initialization and termination logic.

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 Java 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) 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 BEA WebLogic Enterprise software that extend the CORBA APIs:

- Factories and the FactoryFinder object

- Factory-based routing

For information about the FactoryFinder object, see Chapter 5, “FactoryFinder Interface.” For more information about Factory-based routing, see the *Administration Guide*.

Usage Note

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.

Error Conditions and Exceptions

The following paragraphs discuss error conditions and resulting exceptions.

Exceptions Raised by the Java TP Framework

The following exceptions are raised by the Java TP Framework and are returned to clients when error conditions occur in, or are detected by, the Java TP Framework:

```
CORBA.INTERNAL
CORBA.OBJECT_NOT_EXIST
CORBA.OBJ_ADAPTER
CORBA.INVALID_TRANSACTION
CORBA.TRANSACTION_ROLLEDBACK
```

Since the reason for these exceptions may be ambiguous, each time one of these exceptions is raised, the Java TP Framework also writes to the user log file a descriptive error message that explains the reason.

Exceptions in the Server Application Code

The following Java TP Framework callback methods are initiated by events other than client requests on the object:

```
com.beasys.Tobj_ServantBase.activate_object()
com.beasys.Tobj_ServantBase.deactivate_object()
com.beasys.Server.create_servant()
```

If exception conditions are raised in these methods, those exact exceptions are not reported back to the client. However, each of these methods is defined to raise an exception that includes a reason string. The Java TP Framework catches the exception raised by the callback and logs the reason string to the user log file. The Java TP Framework may raise an exception back to the client. Refer to the descriptions of the individual Java TP Framework callback methods for more information about these exceptions.

Example

For `com.beasys.Tobj_ServantBase.deactivate_object()`, the following line of code throws a `DeactivateObjectFailed` exception:

```
throw new com.beasys.TobjS.DeactivateObjectFailed( “deactivate
                                              failed to save state!”);
```

This message is appended to the user log file with a tag made up of the time (hhmmss), system name, process name, and process-id of the calling process. The tag is terminated with a colon. The preceding throw statement causes the following line to appear in the user log file:

```
151104.T1!simpapps.247: APPEXC: deactivate failed to save state!
```

Where `151104` is the time (3:11:04pm), `T1` is the system name, `simpapps` is the process name, `247` is the process-id, and `APPEXC` identifies the message as an application exception message.

Exceptions and Transactions

Exceptions that are raised in either CORBA object methods or in TP Framework callback methods will not automatically cause a transaction to be rolled back unless the TP Framework started the transaction. It is up to the application code to call `Current.rollback_only()` if the condition that caused the exception to be raised should also cause the transaction to be rolled back.

4 Java Bootstrap Object Programming Reference

This topic includes the following sections:

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

Why Bootstrap Objects Are Needed

The Problem: To communicate with BEA WebLogic Enterprise objects, a client application must obtain object references. The client application uses the Bootstrap object to obtain initial object references to six key objects in a BEA WebLogic Enterprise domain:

- FactoryFinder—used to locate factory objects
- SecurityCurrent—used to log on to the system
- TransactionCurrent—used to manage transactions
- InterfaceRepository—used to obtain information about available interfaces
- NotificationService—used to locate Notification Service channel factory objects
- Tobj_SimpleEventsService—used to locate BEA Simple Events Service channel factory objects

However, this poses a problem: *How does the client application access the Bootstrap object?*

The solution: 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 BEA WebLogic Enterprise IIOP Server Listener/Handler. Given this information, the Bootstrap object can generate object references for the above-mentioned remote objects in the BEA WebLogic Enterprise domain. These object references can then be used to access services available in the BEA 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
- InterfaceRepository
- NotificationService
- Tobj_SimpleEventsService

Bootstrap objects may represent the first connection to a specific BEA WebLogic Enterprise domain depending on the format of the IIOP Server Listener/Handler address. If the Null scheme Universal Resource Locator (URL) format is used (the only address format supported in releases of BEA WebLogic Enterprise prior to V5.1), the Bootstrap objects represent the first connection. However, if the URL format is used, the connection will not occur until after Bootstrap object creation. For more information on address formats and connection times, refer to the description of `Tobj_Bootstrap` in the *Java API Reference*, which is included in the Javadoc online documentation.

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

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

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

- SecurityCurrent

  The SecurityCurrent object is used to establish a security context within a BEA 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 BEA 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 BEA 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 BEA 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.

- NotificationService

  The NotificationService object is used to obtain a reference to the event channel factory (CosNotifyChannelAdmin::EventChannelFactory) in the CosNotification Service. In the BEA WebLogic Enterprise system, the EventChannelFactory is used to locate the Notification Service channel.

- Tobj_SimpleEventsService

  The Tobj_SimpleEventsService object is used to obtain a reference to the event channel factory (Tobj_SimpleEvents::ChannelFactory) in the BEA Simple Events Service. In the BEA WebLogic Enterprise system, the ChannelFactory is used to locate the BEA Simple Events Service channel.

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

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

- `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 BEA 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.

Note: Resolving the SecurityCurrent when using the new URL address format (`corbaloc://hostname:port_number`) is a local operation; that is, no connection is made by the client to the IIOP Server Listener/Handler.

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 does not destroy the session with the BEA WebLogic Enterprise domain. After invoking the `destroy_current` method, new instances of the Current objects can be created within the BEA 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 BEA 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 `//host:port`.

Note: 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 CORBA Client Applications* and *Creating CORBA Java Server Applications*.

Note: 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

Client	Description
CORBA C++	CORBA C++ client applications use the BEA WebLogic Enterprise C++ environmental objects to access the CORBA objects in a BEA WebLogic Enterprise domain, and the BEA WebLogic Enterprise Object Request Broker (ORB) to process from CORBA objects. Use the BEA WebLogic Enterprise system development commands to build these client applications (see *Commands, System Processes, and MIB Reference*).
CORBA Java	CORBA Java client applications use the Java environmental objects to access CORBA objects in a BEA WebLogic Enterprise domain. However, these client applications use an ORB product other than the BEA 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 BEA WebLogic Enterprise software supports interoperability with client platforms using of the following: ■ The Java IDL ORB provided with the Java Development Kit 1.2 from Sun Microsystems, Inc. For complete details about Java application and applet support, see the *Release Notes*.
ActiveX	Use the BEA WebLogic Enterprise Automation environmental objects to access CORBA objects in a BEA 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, which are built using a development tool such as Microsoft Visual Basic, Delphi, or PowerBuilder.

This container 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 *CORBA 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 BEA 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 BEA 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 BEA 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* in the Javadoc online documentation. The sections that follow describe:

- The object references returned by the Bootstrap object
- The Java mapping for the Bootstrap object

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`)
NotificationService	EventChannelFactory object (`CosNotifyChannelAdmin.EventChannelFactory`)
Tobj_SimpleEventsService	BEA Simple Events ChannelFactory object (`Tobj_SimpleEvents.ChannelFactory`)

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 BEA 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.
`org.omg.CORBA.BAD_PARAM`	Raised for the `register_callback_port` method if the object is null or if the hostname contained in the object does not match the connection.

Table 4-3 Tobj Module Exceptions (Continued)

Exception	Description
`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_str)
              throws org.omg.CORBA.SystemException;
public class Tobj_Bootstrap {
      public Tobj_Bootstrap(org.omg.CORBA.ORB orb,
                             String address_str,
                             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 the following Java client programming examples that use Bootstrap objects.

- Getting a SecurityCurrent Object
- Getting a UserTransaction Object

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)
      {
               Properties prop = null;
               Tobj.PrincipalAuthenticator auth = null;
               String host_port = "//COLORMAGIC:10000";
               // Set host and port.
               if (args.length == 1) host_port = args[0];
               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

Listing 4-3 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

```
Properties prop = null;
Tobj.PrincipalAuthenticator auth = null;
String host_port = "//COLORMAGIC:10000";
// Set host and port.
if (args.length == 1) host_port = args[0];
try {
// 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();
```

5 FactoryFinder Interface

This topic includes the following sections:

- Capabilities, Limitations, and Requirements
- Functional Description. This section describes:
  - 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 BEA WebLogic Enterprise domain. The BEA 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 BEA 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 BEA 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 BEA 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 BEA 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 BEA 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

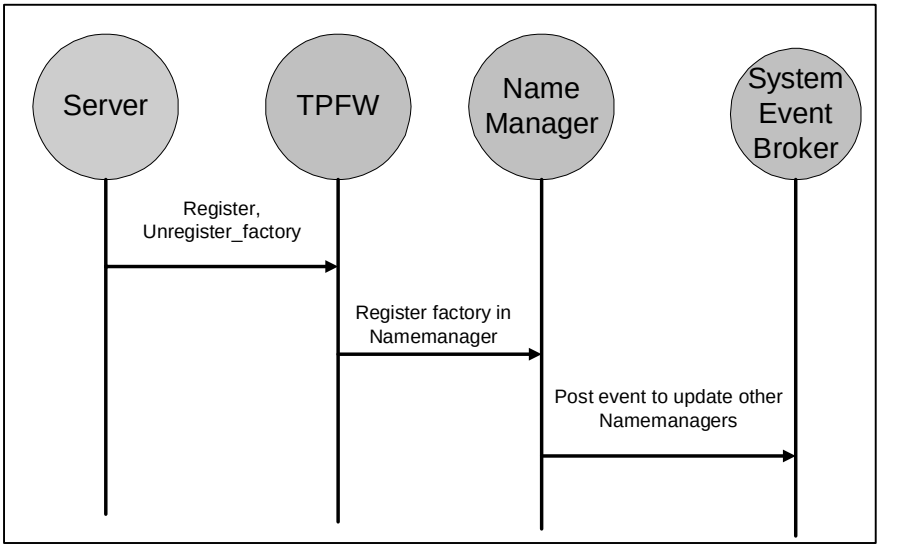
"FactoryFinder". This operation returns a reference to a FactoryFinder that is in the domain to which the client application is currently attached. For more information, see the description of the com.beasys.Tobj_Bootstrap object in API Javadoc.

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 BEA 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

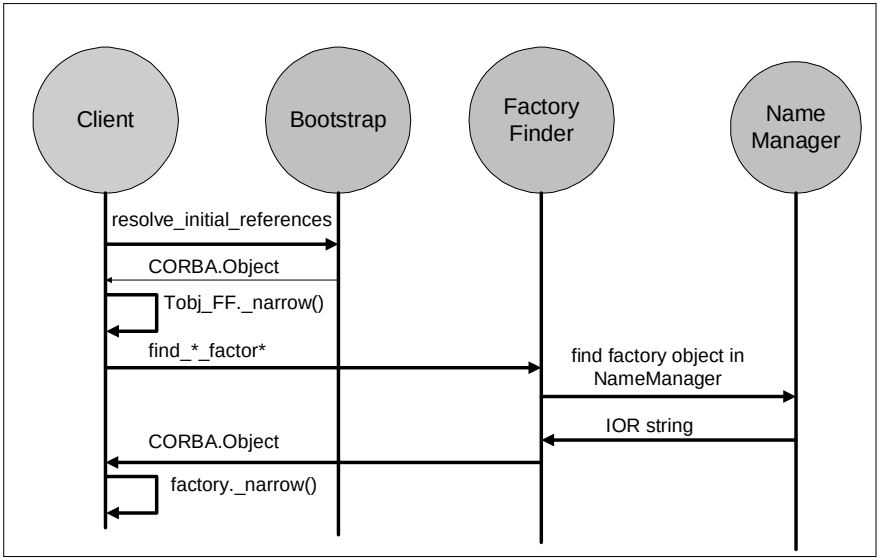


When the server application is shutting down, it must unregister any references to the factory object that it has previously registered in the application server. This is done by passing the same reference to the factory object, along with the corresponding value used to identify the factory, to the `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 “Java TP Framework Interfaces” on page 3-18.

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 BEA 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 BEA 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 BEA 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)
```

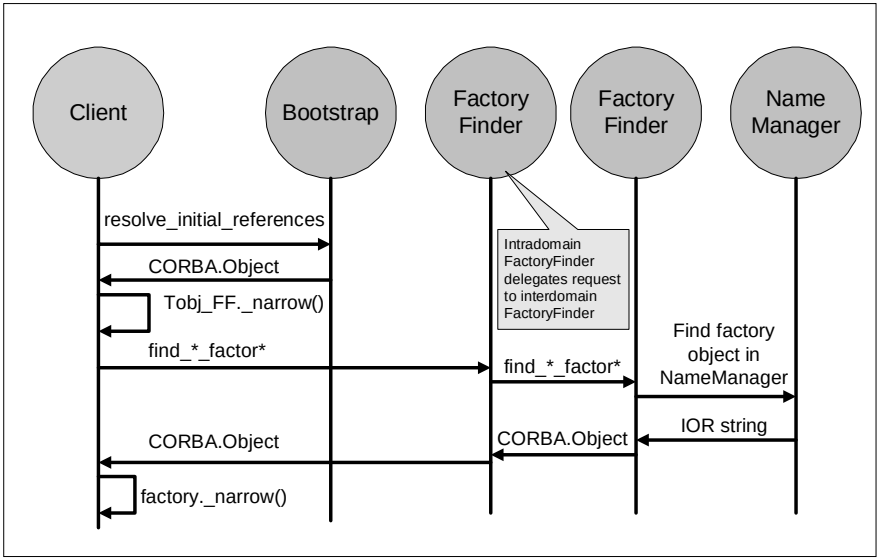
```
                raises (CosLifeCycle::NoFactory,
                        CannotProceed,
                        RegistrarNotAvailable);
      CosLifeCycle::Factory find_one_factory_by_id(in string
                                                   factory_id)
                 raises (CosLifeCycle::NoFactory,
                         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 BEA WebLogic Enterprise Extensions?

The BEA 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, BEA 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 i^{th} 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 i^{th} 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

For a detailed discussion of Securty, see *Using Security*. This document provides an introduction to crytography and other concepts associated with the BEA WebLogic Enterprise security features, a description of how to secure your applications using the BEA WebLogic Enterprise security features, and a guide to the use of the application programming interfaces (APIs) in the BEA WebLogic Enterprise Security Service.

A PDF file of *Using Security* is also provided in the online documentation.

CHAPTER

7 Transactions Service

For a detailed discussion of Transactions, see *Using Transactions*. This document provides an introduction to transactions, a description the application programming interfaces (APIs), and a guide to the use of the application programming interfaces (APIs) to develop applications.

A PDF file of *Using Transactions* is also provided in the online documentation.

8 Notification Service

For a detailed discussion of the Notification Service, see *Using the Notification Service*. This document provides an introduction to the Notification Service, a description the application programming interfaces (APIs), and a guide to the use of the application programming interfaces (APIs) to develop applications.

A PDF file of *Using the Notification Service* is also provided in the online documentation.

9 Request-Level Interceptors

For a detailed discussion of request-level interceptors, see *Using Request-Level Interceptors*. This document provides an introduction to request-level interceptors, a description the application programming interfaces (APIs), and a guide to the use of the application programming interfaces (APIs) to implement request-level interceptors.

A PDF file of Using *Request-Level Interceptors* is also provided in the online documentation.

CHAPTER

10 Interface Repository Interfaces

This topic includes the following sections:

- Structure and Usage
- Building Client Applications
- Getting Initial References to the InterfaceRepository Object
- Interface Repository Interfaces. This section describes:
  - 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 BEA WebLogic Enterprise implementation of the Interface Repository interfaces. Used with permission by OMG.

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

The BEA 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 BEA 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 *Commands, System Processes, and MIB Reference*.

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 BEA 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 *Commands, System Processes, and MIB Reference*. 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 CORBA 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 10-5. 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, "Java Bootstrap Object Programming Reference.") 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 BEA WebLogic Enterprise ORB, the BEA 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, "Java Bootstrap Object Programming Reference."

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 BEA 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 BEA 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

11 Joint Client/Server Applications

This topic includes the following sections:

- Introduction. This section describes:
  - 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 API Javadoc.

Introduction

For either a BEA 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 BEA WebLogic Enterprise environmental objects to establish connections, set up security, and start transactions.

BEA 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 BEA WebLogic Enterprise environmental objects that are supplied with the BEA WebLogic Enterprise software to perform functions unique to BEA WebLogic Enterprise.

For BEA WebLogic Enterprise joint client/server applications, the client code must be structured so that it can act as a server for callback BEA WebLogic Enterprise objects only. Such clients do not use the TP Framework and are not subject to BEA WebLogic Enterprise system administration. Besides the programming implications, this means that joint client/server applications do not have the same scalability and reliability as BEA 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 BEA WebLogic Enterprise server, rather than in a client.

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

Main Program and Server Initialization

In a BEA 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 BEA 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 BEA WebLogic Enterprise joint client/server application (as for a BEA 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 11-3.

Servants

Servants (method code) for BEA WebLogic Enterprise joint client/server applications are very similar to servants for BEA 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 BEA 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 11-4 and "Preparing Callback Objects Using BEAWrapper Callbacks" on page 11-6.

Servant Inheritance from Skeletons

In a BEA WebLogic Enterprise client, as well as in a BEA WebLogic Enterprise 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 BEA WebLogic Enterprise 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 BEA WebLogic Enterprise 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

BEA 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 BEA 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 BEA WebLogic Enterprise 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, BEA WebLogic Enterprise 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 BEA WebLogic Enterprise 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 BEA WebLogic Enterprise 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 BEA WebLogic Enterprise application.

Port Numbers for Persistent Object References

BEA WebLogic Enterprise Java server applications support only GIOP 1.0, as described in Chapter 13 of the OMG CORBA 2.2 specification.

For a BEA WebLogic Enterprise 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 BEA WebLogic Enterprise 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 API Javadoc.

CHAPTER

12 Java Development and Administration Commands

For a detailed discussion of BEA WebLogic Enterprise development and administrative commands, see *Commands, System Processes, and MIB Reference*. This document describes all BEA WebLogic Enterprise commands and processes.

A PDF file of *Commands, Processes, and MIB Reference* is also provided in the online documentation.

13 CORBA ORB

This chapter supplements the information in package `org.omg.CORBA` by providing information on the following topics:

- Initializing the ORB
- Passing the Address of the IIOP Listener

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 BEA 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 BEA 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 BEA 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.

14 Mapping IDL-to-Java

This topic includes the following sections:

- IDL-to-Java Overview
- Package Comments on Holder Classes
- Exceptions. This section describes:
  - 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 BEA WebLogic Enterprise system. When compiling the OMG IDL files to build server skeletons to be used with the BEA 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 BEA 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” on page 14-7.

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 14-1 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 14-2 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.