BEA Tuxedo®

Using the CORBA idltojava Compiler

Release 8.1
January 2003

Contents

About This Document

What You Need to Know vi
e-docs Web Site vi
How to Print the Document vi
Related Information vii
Contact Us! vii
Documentation Conventions viii

1. Overview of idltojava Compiler

Where Do I Get the BEA idltojava Compiler? 1-2
How Does the BEA idltojava Compiler Differ from the Sun Microsystems, Inc. Version? 1-2
What Is IDL? 1-3
What Is Java IDL? 1-4
Accessing CORBA Objects 1-4

2. Using the idltojava Command

Syntax of the idltojava Command 2-2
idltojava Command Description 2-2
Running idltojava on Client or Joint Client/Server IDL Files 2-2
idltojava Command Options 2-3
idltojava Command Flags 2-4
Using #pragma in IDL Files 2-6

3. Java IDL Examples

Getting Started with a Simple Example of IDL 3-1
Callback Objects IDL Example 3-2

4. Java IDL Programming Concepts

Exceptions 4-1
- Differences Between CORBA and Java Exceptions 4-2
- System Exceptions 4-2
  - System Exception Structure 4-2
  - Completion Status 4-3
- User Exceptions 4-3
- Minor Code Meanings 4-4

Initializations 4-7
- Creating an ORB Object 4-8
  - Creating an ORB for an Application 4-8
  - Creating an ORB for an Applet 4-8
  - Arguments to ORB.init() 4-9
  - System Properties 4-10
- Obtaining Initial Object References 4-10
  - Stringified Object References 4-11
  - Getting References from the ORB 4-11

The FactoryFinder Interface 4-12

5. IDL-to-Java Mappings Used By the idltojava Compiler

Index

About This Document

This document explains what Java Interface Definition Language (IDL) is and describes how to use the idltojava compiler for developing CORBA Java clients and CORBA Java joint client/servers in the BEA Tuxedo® environment. CORBA Java clients and CORBA Java joint client/servers communicate with the Application-to-Transaction Monitor Interface (ATMI) environment in the BEA Tuxedo product using the Internet Inter-ORB Protocol (IIOP). The ATMI environment in the BEA product does not support the hosting of CORBA Java server objects.

This document includes the following topics:

- Chapter 1, "Overview of idltojava Compiler," explains the relationship of Java IDL to CORBA, and explains how you can use Java IDL to create CORBA Java clients and CORBA Java joint client/servers that interoperate with CORBA objects. This chapter also explains where to get the BEA idltojava compiler, and how the BEA idltojava compiler differs from the idltojava compiler available from Sun Microsystems, Inc.
- Chapter 2, "Using the idltojava Command," explains how to run the idltojava compiler and explains all the options and flags on the `idltojava` command.
- Chapter 3, "Java IDL Examples," provides several code examples to illustrate the use of the idltojava compiler. The code examples include the Java SimpApp sample application to get you started. The other example included in this topic illustrates the use of Callback Objects.
- Chapter 4, "Java IDL Programming Concepts," discusses some relevant programming concepts, such as Exceptions, Initialization, and use of the Factory Finder object.
- Chapter 5, "IDL-to-Java Mappings Used By the idltojava Compiler," explains the CORBA IDL-to-Java mappings that the idltojava compiler implements.

What You Need to Know

This document is intended mainly for developers who are interested in building CORBA Java clients and CORBA Java joint client/servers that can interact with CORBA objects. It assumes a familiarity with the CORBA programming environment and Java programming.

e-docs Web Site

The BEA Tuxedo 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 BEA Tuxedo 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 BEA Tuxedo 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, BEA Tuxedo, distributed object computing, transaction processing, C++ programming, and Java programming, see the *CORBA Bibliography* in the BEA Tuxedo online documentation.

For more general information about Java IDL and Java CORBA applications, refer to the following sources:

- The Object Management Group (OMG) Web site at http://www.omg.org/
- The Sun Microsystems, Inc. Java Web site at http://java.sun.com/

Contact Us!

Your feedback on the BEA Tuxedo 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 BEA Tuxedo documentation.

In your e-mail message, please indicate that you are using the documentation for the BEA Tuxedo 8.0 release.

If you have any questions about this version of BEA Tuxedo, or if you have problems installing and running BEA Tuxedo 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.
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`*

Convention	Item
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Examples*: LPT1 SIGNON OR
`{ }`	Indicates a set of choices in a syntax line. The braces themselves should never be typed.

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

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

`...`	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.

CHAPTER

1 Overview of idltojava Compiler

The CORBA environment in the BEA Tuxedo product allows CORBA Java clients and CORBA Java joint client/servers to invoke operations on CORBA objects in a BEA Tuxedo domain using the industry standard Object Management Group (OMG) Interface Definition Language (IDL) and Internet Inter-ORBProtocol (IIOP) defined by the OMG.

To build CORBA Java clients and CORBA Java joint client/servers that can access CORBA objects, you need the BEA idltojava compiler, a tool that converts OMG IDL files to Java client stub and skeleton files. The idltojava compiler is included with the BEA Tuxedo software.

This topic includes the following sections:

- Where Do I Get the BEA idltojava Compiler?
- How Does the BEA idltojava Compiler Differ from the Sun Microsystems, Inc. Version?
- What Is IDL?
- What Is Java IDL?
- Accessing CORBA Objects

Where Do I Get the BEA idltojava Compiler?

The BEA Tuxedo CD-ROM includes the BEA version of the idltojava compiler. Once you have installed the BEA Tuxedo software, you can find the idltojava compiler in `TUXDIR/bin`.

How Does the BEA idltojava Compiler Differ from the Sun Microsystems, Inc. Version?

The idltojava compiler provided with the BEA Tuxedo product includes several enhancements, extensions, and additions that are not included in the original compiler produced by Sun Microsystems, Inc. This topic includes a summary of the revisions in the compiler included with the BEA Tuxedo product. For detailed information on using the idltojava compiler provided with the BEA Tuxedo product, see the topic Using the idltojava Command.

The BEA Tuxedo idltojava compiler:

- The behavior and defaults of the flags differ from that described in the Sun Microsystems, Inc. documentation. (See idltojava Command Flags.)
- Includes a new #pragma tag: `#pragma ID` *`<name>`* *`<Repostitory_id>`* (See Using #pragma in IDL Files.)
- Includes a new #pragma tag: `#pragma version` *`<name>`* *`<m.n>`* (See Using #pragma in IDL Files.)
- Extends the `#pragma prefix` to work on inner scope. A blank prefix reverts. (See Using #pragma in IDL Files.)
- Allows unions with Boolean discriminators.
- Allows declarations nested inside complex types.

What Is IDL?

Interface Definition Language (IDL) is a generic term for a language that lets a program or object written in one language communicate with another program written in an unknown language. In distributed object technology, new objects must be able to be sent to any platform environment and have the ability to discover how to run in that environment. An ORB is an example of a program that uses an interface definition language to "broker" communication between one object program and another.

All CORBA objects support an IDL interface; the IDL interface defines an *object type*. An interface can inherit from one or more other interfaces. IDL syntax is very similar to that of Java or C++, and an IDL file is functionally the CORBA language-independent equivalent to a C++ header file. IDL is mapped into each programming language to provide access to object interfaces from that language. With Java IDL, these IDL interfaces can be translated to Java using the idltojava compiler. For each IDL interface, the idltojava compiler generates a Java interface and the other `.java` files needed, including a client stub and a server skeleton.

An IDL interface declares a set of client-accessible operations, exceptions, and typed attributes (values). Each operation has a signature that defines its name, parameters, result, and exceptions. Listing 1-1 shows the IDL for the Simple interface in the Simpapp sample application included with the BEA Tuxedo product.

Listing 1-1 An IDL Interface for the Simpapp Sample Application

```
#pragma prefix "beasys.com"

interface Simple
{
     //Convert a string to lower case (return a new string)
     string to_lower(in     string val);

     //Convert a string to upper case (in place)
     void to_upper(inout string val);
};

interface SimpleFactory
{
     Simple find_simple();
};
```

What Is Java IDL?

Java IDL is not a particular kind of Interface Definition Language (IDL) apart from OMG IDL. The same IDL can be compiled with the idltojava compiler to produce CORBA-compatible Java files, or with a C++ based compiler to produce CORBA-compatible C++ files. The compiler that you use on the IDL is what makes the difference. The OMG has established IDL-to-Java mappings as well as IDL-to-C++ mappings. The language-based compilers generate code based on the OMG CORBA mappings to their particular language.

The BEA Tuxedo product provides its own "brand" of Java IDL. In other words, the BEA Tuxedo product provides all of the components you need to build CORBA Java clients and CORBA Java joint client/servers that are capable of accessing CORBA objects. The key components in the BEA Tuxedo product are listed in Accessing CORBA Objects.

Accessing CORBA Objects

You can build two types of applications using the idltojava compiler:

- A CORBA Java client which uses files from the `idltojava` command for its client stubs.
- A CORBA Java joint client/server which uses files from the `idltojava` command for its client stubs and its server skeletons.

CORBA Java clients and CORBA Java joint client/servers communicate with the Application-to-Transaction Monitor Interface (ATMI) environment in the BEA Tuxedo product using the Internet Inter-ORB Protocol (IIOP). The ATMI environment in the BEA product does not support the hosting of CORBA Java server objects.

Note: A *joint client/server* is a client that implements CORBA server objects to be used as callback objects. The server role of the remote joint client/server is considerably less robust than that of a BEA Tuxedo CORBA server. For more information about joint client/servers, see *Using CORBA Server-to-Server Communication*.

The BEA Tuxedo product provides all of the components you need to build CORBA Java clients and CORBA Java joint client/servers capable of accessing CORBA objects. The key components are:

- idltojava compiler—a tool for converting IDL interface definitions to Java client stubs and skeleton files. The `idltojava` command compiles standard CORBA IDL source code into Java source code. (You can then use the `javac` compiler to compile that source to Java bytecodes.) For a detailed description of the idltojava compiler, see Using the idltojava Command
- Bootstrap Object and FactoryFinder—CORBA objects that supply local and remote object references. A CORBA Java client or CORBA Java joint client/server uses the Bootstrap object to obtain initial object references to CORBA objects in a BEA Tuxedo domain, one of which is the FactoryFinder.

CORBA Java clients and CORBA Java joint client/servers obtain object references from:

- A factory object

  For example, the client could invoke a `create` method on `DocumentFactory` object to create a new `Document`. The `DocumentFactory` `create` method would return an object reference for `Document` to the client.

  The use of a factory object to obtain object references is the recommended method for CORBA Java clients in this release of the BEA Tuxedo product.

  The FactoryFinder object registers, stores, and finds CORBA objects within a single BEA Tuxedo domain or across multiple domains.

  For detailed information on how to use the FactoryFinder object, see the *CORBA Javadoc*.
- A string that was specially created from an object reference

After an object reference is obtained, the client must *narrow* it to the appropriate type. IDL supports inheritance; the root of its hierarchy is `Object` in IDL, `org.omg.CORBA.Object` in Java. (`org.omg.CORBA.Object` is, of course, a subclass of `java.lang.Object`.) Some operations, notably name lookup and unstringifying, return an `org.omg.CORBA.Object`, which you narrow (using a helper class generated by the idltojava compiler) to the derived type you want the object to be. CORBA objects must be explicitly narrowed because the Java run time cannot always know the exact type of a CORBA object.

CHAPTER

2 Using the idltojava Command

The idltojava compiler compiles IDL files to Java source code based on IDL-to-Java mappings defined by the OMG. For more information about the IDL-to-Java mappings, refer to the topic IDL-to-Java Mappings Used By the idltojava Compiler.

This topic includes the following sections:

- Syntax of the idltojava Command
- idltojava Command Description
- Running idltojava on Client or Joint Client/Server IDL Files
- idltojava Command Options
- idltojava Command Flags
- Using #pragma in IDL Files

For a quick summary of the enhancements and updates added to the BEA Tuxedo idltojava compiler, see the topic Overview of idltojava Compiler.

Syntax of the idltojava Command

The following is an example of the `idltojava` command syntax:

```
idltojava [idltojava Command Flags] [idltojava Command Options] filename ...
```

idltojava Command Description

The `idltojava` command compiles IDL source code into Java source code. You then use the `javac` compiler to compile that source to Java bytecodes.The command `idltojava` is used to translate IDL source code into generic client stubs and generic server skeletons which can be used for callbacks.

The IDL declarations from the named IDL files are translated to Java declarations according to the mappings specified in the OMG IDL-to-Java mappings. (For more information on the mappings, see IDL-to-Java Mappings Used By the idltojava Compiler.)

Running idltojava on Client or Joint Client/Server IDL Files

To run `idltojava` on client-side IDL files for either CORBA Java clients or CORBA Java joint client/servers, use the following command:

```
idltojava <flags> <options> <idl-files>
```

The `idltojava` command requires a C++ preprocessor, and is used to generate deprecated names. The command `idltojava` generates Java code as is appropriate for the Java client ORB.

Note: A remote *joint client/server* is a client that implements server objects to be used as callback objects. The server role of the remote joint client/server is considerably less robust than that of a BEA Tuxedo server. For more information about joint client/servers, see *Using CORBA Server-to-Server Communication*.

idltojava Command Options

Note: Several option descriptions have been added here that are not documented in the original Sun Microsystems, Inc. idltojava compiler documentation (see Table 2-1).

Table 2-1 idltojava Added Options

Option	Description
`-j javaDirectory`	Specifies that generated Java files should be written to the given *directory*. This directory is independent of the `-p` option, if any.
`-J filesFile`	Specifies that a list of the files generated by **idltojava** should be written to *filesFile*.
`-p package-name`	Specifies the name of an outer package to enclose all the generated Java files. It has the same function as `#pragma javaPackage`. **Note:** You must include an *outer package*. The compiler does not do this for you. If you do not have an outer package, the idltojava compiler will still generate Java files for you but you will get a Java compiler error when you try to compile the `*.java` files.
The following options are identical to the equivalent C/C++ compiler options (cpp):	
`-Idirectory`	Specifies a directory or path to be searched for files that are *#included* in IDL files. This option is passed to the preprocessor.

Table 2-1 idltojava Added Options (Continued)

Option	Description
`-Dsymbol`	Specifies a symbol to be defined during preprocessing of the IDL files. This option is passed to the preprocessor.
`-Usymbol`	Specifies a symbol to be undefined during preprocessing of the IDL files. This option is passed to the preprocessor.

idltojava Command Flags

The flags can be turned on by specifying them as shown, and they can be turned off by prefixing them with the letters **`no-`**. For example, to prevent the C preprocessor from being run on the input IDL files, use **`-fno-cpp`**.

Table 2-2 includes descriptions of all flags.

Table 2-2 idltojava Command Flags

Flag	Description
`-flist-flags`	Requests that the state of all the **`-f`** flags be printed. The default value of this flag is `off`.
`-flist -debug-flags`	Provides a list of debugger flags.
`-fcaseless`	Requests that the use of upper- and lowercase letters in keywords and identifiers not be significant. Note that this does *not* mean that case is ignored, because all uses of an identifier must have the same use of case as the initial usage. For example, “Session” and “session” are the same identifier, but using “session” after an initial use of “Session” results in an error because “session” does not have the case as “Session.” CORBA uses this definition of caseless to allow accurate mappings to case-sensitive languages. The default value of this flag is on. The severity of identifier conflicts found with this flag specified is warning (the default). See the `-fstrict` flag for more information.
`-fclient`	Requests the generation of the client side of the IDL files supplied. The default value of this flag is `on`.

Table 2-2 idltojava Command Flags (Continued)

Flag	Description
`-fcpp`	Requests that the IDL source be run through the C/C++ preprocessor before being compiled by the idltojava compiler. The default value of this flag is `on`.
`-fignore-duplicates`	Specifies that duplicate definitions be ignored. This may be useful if compiling multiple IDL files at one time. The default value of this flag is `off`.
`-flist-options`	Lists the options specified on the command line. The default value of this flag is `off`.
`-fmap-included-files`	Specifies that Java files be generated for definitions included by `#include` preprocessor directives. The default value for this flag is `off`, which specifies that the Java files for included definitions **not** be generated.
`-fserver`	Requests the generation of the server side of the IDL files supplied. The default value of this flag is `on`.
`-fverbose`	Requests that the compiler comment on the progress of the compilation. The default value of this flag is `off`.
`-fversion`	Requests that the compiler print its version and timestamp. The default value of this flag is `off`.
`-fwarn-pragma`	Requests that warning messages be issued for unknown or improperly specified `#pragmas`. The default value of this flag is `on`.
`-fwrite-files`	Requests that the derived Java files be written. The default value of this flag is `on`. You might specify `-fno-write-files` if you wished to check for errors without actually writing the files.
`-fstrict`	In previous releases, the severity of identifier conflicts found with the `-fcaseless` flag specified was `error`. In this release, the `-fcaseless` flag performs its identifier comparisons exactly as before but the severity of the conflicts has changed from `error` to `warning`. To request the previous, strict CORBA conformance which requires identifier conflicts to be specified as `error`, specify this flag with the `-fcaseless` flag.

Using #pragma in IDL Files

Note: The BEA Tuxedo idltojava compiler processes `#pragma` somewhat differently from the Sun Microsystems, Inc. idltojava compiler.

`RepositoryPrefix``="prefix"`

A default repository prefix can also be requested with the line `#pragma prefix "requested prefix"` at the top-level in the IDL file itself. The line:

```
#pragma javaPackage "package"
```

wraps the default package in one called package. For example, compiling an IDL module `M` normally creates a Java package `M`. If the module declaration is preceded by:

```
#pragma javaPackage browser
```

the compiler will create the package `M` inside package browser. This pragma is useful when the definitions in one IDL module will be used in multiple products. The command-line option `-p` can be used to achieve the same result. The line:

```
#pragma ID scoped-name "IDL:<path>:<version>"
```

specifies the repository ID of the identifier scoped-name. This pragma may appear anywhere in an IDL file. If the pragma appears inside a complex type, such as structure or union, then only as much of scoped-name need be specified to specify the element. A scoped-name is of the form `outer_name::name::inner_name`. The `<path>`component of the repository ID is a series of identifiers separated by forward slashes `(/)`. The `<version>` component is a decimal number `MM.mm`, where `MM` is the major version number and `mm` is the minor version number.

CHAPTER

3 Java IDL Examples

This topic includes the following sections:

- Getting Started with a Simple Example of IDL
- Callback Objects IDL Example

Getting Started with a Simple Example of IDL

Listing 3-1 shows the OMG IDL to describe a CORBA object whose operations `to_lower()` and `to_upper()` each return a single string in which the letter case of the user input is changed accordingly. (Uppercase input is changed to lowercase, and vice versa.)

Listing 3-1 IDL Interface for the Simpapp Sample Application

```
#pragma prefix "beasys.com"

interface Simple
{
     //Convert a string to lower case (return a new string)
     string to_lower(in     string val);

     //Convert a string to upper case (in place)
     void to_upper(inout string val);
};
```

```
interface SimpleFactory
{
     Simple find_simple();
};
```

If you were implementing this application from scratch, you would compile this IDL interface with the following command:

```
idltojava Simple.idl
```

This would generate stubs and skeletons and several other files.

For information on the options and flags on the idltojava compiler, refer to the topic Using the idltojava Command.

Callback Objects IDL Example

Listing 3-2 shows the OMG IDL to define the Callback, Simple, and SimpleFactory interfaces in the Callback sample application.

Listing 3-2 IDL Definition for the Callback Sample Application

```
#pragma prefix "beasys.com"

     interface Callback

    //This method prints the passed data in uppercase and lowercase
     //letters.
     {
          void print_converted(in string message);
     };

     interface Simple

     //Call the callback object in the joint client/server
     //application
     {
          void call_callback(in string val, in Callback
                                        callback_ref);
};

interface SimpleFactory
{
     Simple find_simple();
};
```

For a complete explanation of the Java CORBA callbacks example as well as information on how to build and run the example, see *Using CORBA Server-to-Server Communication* in the BEA Tuxedo online documentation.

4 Java IDL Programming Concepts

This topic includes the following sections:

- Exceptions
- Initializations
- The FactoryFinder Interface

Exceptions

CORBA has two types of exceptions: standard system exceptions, which are fully specified by the OMG, and user exceptions, which are defined by the individual application programmer. CORBA exceptions differ slightly 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
- 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 need not (and 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 including:

- Server-side system exceptions, such as resource exhaustion or activation failure.
- Communication system exceptions, for example, losing contact with the object, host down, or cannot talk to the ISL or ISH.
- 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 Java client should always catch CORBA system exceptions. Moreover, developers cannot rely on the idltojava 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.
```

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

Every system exception has a "minor" field that allows CORBA vendors to provide additional information about the cause of the exception. Table 4-1 and Table 4-2 list the minor codes of Java IDL's system exceptions and describes their significance.

Table 4-1 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	
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.

Table 4-1 ORB Minor Codes and Their Meanings (Continued)

Code	Meaning
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 the incorrect length.
7	Unable to determine the local hostname using the Java APIs `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 1.0 message type found.
14	Error encountered while unmarshaling the user exception.
18	Internal initialization error.
INV_OBJREF Exception Minor Codes	
1	An IOR with no profile was encountered.
MARSHAL Exception Minor Codes	
4	Error occurred 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.

Table 4-1 ORB Minor Codes and Their Meanings (Continued)

Code	Meaning
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 occurred when trying to connect a servant to the ORB.
OBJ_NOT_EXIST Exception Minor Codes	
1	Locate request received 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 contents 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.
3	Unknown run-time exception thrown by the server implementation.

Table 4-2 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.

Table 4-2 Name Server Minor Codes and Their Meanings (Continued)

Code	Meaning
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	Implementation of the destroy operation caught a Java exception while disconnecting from the ORB.

Initializations

Before a CORBA Java client or a CORBA Java joint client/server can use CORBA objects, it must initialize itself by:

- Creating an ORB object.
- Obtaining one or more initial object references, typically using a FactoryFinder object.

Creating an ORB Object

Before it can create or invoke a CORBA object, a CORBA Java client or a CORBA Java joint client/server must first create an ORB object. By creating an ORB object, the client or joint client/server introduces itself to the ORB and obtains access to important operations that are defined on the ORB object.

Clients and joint client/servers create ORB instances slightly differently, because their parameters, which must be passed in the `ORB.init()` call, are arranged differently.

Creating an ORB for an Application

The following code fragment shows how a CORBA Java client might create an ORB:

```
import org.omg.CORBA.ORB;

public static void main(String args[])
{
  try{
    ORB orb = ORB.init(args, null);
// code continues
```

Creating an ORB for an Applet

A Java applet creates an ORB like this:

```
import org.omg.CORBA.ORB;

public void init() {
  try {
    ORB orb = ORB.init(this, null);
// code continues
```

Some Web browsers have a built-in ORB. This can cause problems if that ORB is not entirely compliant. In this case, special steps must be taken to initialize the Java IDL ORB specifically. For example, because of missing classes in the installed ORB in Netscape Communicator 4.01, an applet displayed in that browser must contain code similar to the following in its `init()` method:

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

public class MyApplet extends java.applet.Applet {
    public void init()
    {
      // Instantiate the Java IDL ORB, passing in this applet
      // so that the ORB can retrieve the applet properties.
      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);
      ...
    }
  }
```

Arguments to ORB.init()

For both applications and applets, the arguments for the initialization method are:

- `args` or `this`

  Provides the ORB access to the application's arguments or applet's parameters.

- `null`

  A `java.util.Properties` object.

The `init()` operation uses these parameters, as well as the system properties, to obtain information it needs to configure the ORB. It searches for ORB configuration properties in the following places and order:

1. The application or applet parameters (first argument).
2. A `java.util.Properties` object (second argument), if one has been supplied.
3. The `java.util.Properties` object returned by `System.getProperties()`.

The first value found for a particular property is the value used by the `init()` operation. If a configuration property cannot be found in any of these places, the `init()` operation assumes an implementation-specific value for it. For maximum portability among ORB implementations, applets and applications should explicitly specify configuration property values that affect their operation, rather than relying on the assumptions of the ORB in which they are running.

System Properties

The BEA Tuxedo product uses the Sun Microsystem, Inc. Java virtual machine, which adds `-D` command-line arguments to it. Other Java virtual machines may or may not do the same.

Currently, the following configuration properties are defined for all ORB implementations:

- `org.omg.CORBA.ORBClass`

  The name of a Java class that implements the `org.omg.CORBA.ORB` interface. Applets and applications do not need to supply this property unless they must have a particular ORB implementation. The value for the Java IDL ORB is `com.sun.CORBA.iiop.ORB`.

- `org.omg.CORBA.ORBSingletonClass`

  The name of a Java class that implements the `org.omg.CORBA.ORB` interface. This is the object returned by a call to `orb.init()` with no arguments. It is used primarily to create typecode instances than can be shared across untrusted code (such as unsigned applets) in a secured environment.

Applet parameters should specify the full property names. The conventions for applications differ from applets so as not to expose language-specific details in command-line invocations.

Obtaining Initial Object References

To invoke a CORBA object, an applet or application must have a reference for it. There are three ways to get a reference for a CORBA object:

- From a string that was specially created from an object reference
- From another object, such as a FactoryFinder
- From the `bootstrap` method

Stringified Object References

The first technique, converting a stringified reference to an actual object reference, is ORB-implementation independent. Regardless of which ORB an applet or application runs on, it can convert a stringified object reference. However, it is up to the applet or application developer to:

- Ensure that the object referred to is actually accessible from where the applet or application is running.
- Obtain the stringified reference, perhaps from a file or a parameter.

Getting References from the ORB

If you do not use a stringified reference to get an initial CORBA object, you use the ORB itself to produce an initial object reference.

The Bootstrap object defines an operation called `resolve_initial_references()` that is intended for bootstrapping object references into a newly started application or applet. The operation takes a string argument that names one of a few recognized objects; it returns a CORBA Object, which must be narrowed to the type the applet or application knows it to be.

Using the Bootstrap object, you can obtain the following object references (SecurityCurrent, TransactionCurrent, FactoryFinder, NotificationService, Tobj_SimpleEventsService, NameService, and InterfaceRepository). The object of concern to us here is the FactoryFinder object.

The FactoryFinder interface provides clients with one object reference that serves as the single point of entry into the BEA Tuxedo domain. The FactoryFinder object is used to obtain a specific factory object, which in turn can create the needed objects.

For more information on how to use the Bootstrap object, see the *CORBA Programming Reference*.

The FactoryFinder Interface

The FactoryFinder interface provides clients with one object reference that serves as the single point of entry into the BEA Tuxedo domain. Multiple FactoryFinders provide increased availability and reliability. Mapping across multiple domains is supported.

The FactoryFinder interface and the NameManager are a mechanism for registering, storing, and finding objects. In the BEA Tuxedo environment, you can:

- Use the Bootstrap object to obtain an object reference to a FactoryFinder object.
- Use the FactoryFinder object to find the Factory object you need.
- Use the Factory object to create new instances of the CORBA object.

For more information about how to use the FactoryFinder object, see the *CORBA Programming Reference*.

CHAPTER

5 IDL-to-Java Mappings Used By the idltojava Compiler

The idltojava compiler reads an OMG IDL interface and translates or maps it to a Java interface. The idltojava compiler also creates stub, skeleton, helper, holder, and other files as necessary. These `.java` files are generated from the IDL file according to the mapping specified in the OMG document IDL/Java Language Mapping.

For more information on the IDL-to-Java mappings, refer to the OMG Web site at http://www.omg.org.

CORBA objects are defined in OMG IDL (Object Management Group Interface Definition Language). Before they can be used by a Java developer, their interfaces must be mapped to Java classes and interfaces. The idltojava compiler performs this mapping automatically.

Table 5-1 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 data type of its return value, the data types 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.

Table 5-1 IDL Constructs Mapped to Java Constructs

IDL Construct	Java Construct
module	package
interface	interface, helper class, holder class
constant	public static final
boolean	boolean
char, wchar	char
octet	byte
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.

Index

Symbols

#pragma, using in IDL files 2-6

B

Bootstrap object 1-5

C

CORBA
- exceptions in 4-1

D

documentation, where to find it vi

E

exceptions 4-1
- completion status in 4-3
- minor codes in 4-4
- system 4-2
- user 4-3

F

FactoryFinder 1-5
FactoryFinder interface 4-12

I

IDL
- See Interface Definition Language 1-2

IDL interface 1-3
idltojava command
- flags 2-4
- options 2-3
- syntax of 2-2
- using 2-1

idltojava compiler
- differences from Sun version 1-2
- where to get it 1-2

initialization
- of Java program 4-7

interface
- FactoryFinder 4-12
- IDL 1-3

Interface Definition Language (IDL)
- what it is 1-3

J

Java IDL
- examples of 3-1
- what it is 1-3

M

mappings, IDL-to-Java 5-1
minor codes, meaning of 4-4

O

object references

obtaining 1-5, 4-10
ORB object, creating 4-8
ORB.init 4-9

P

pragma, using in IDL files 2-6
printing product documentation vi

S

Sun Microsystems, Inc.
differences between Sun and BEA
idltojava compilers 1-2
support
technical vii